

# *Relazione homework 4*

Luca Mastrobattista

Matricola: 0292461

## Indice

<b>1</b>	<b>Traccia dell'homework</b>	<b>4</b>
1.1	Testo . . . . .	4
1.2	Scadenza . . . . .	4
1.3	Consegna . . . . .	4
<b>2</b>	<b>Ambiente di lavoro</b>	<b>5</b>
2.0.1	Riepilogo risultati dell'import . . . . .	7
2.0.2	Informazioni aggiuntive . . . . .	8
<b>3</b>	<b>Analisi</b>	<b>10</b>
3.1	Basic static (code) analysis . . . . .	10
3.2	Basic dynamic (behaviour) analysis . . . . .	10
3.3	Advanced analysis . . . . .	10
3.3.1	Ricerca dell'OEP . . . . .	11
3.3.2	Fare il <i>dump</i> . . . . .	11
3.4	Advanced analysis - dump-method1.exe . . . . .	12
3.4.1	Primo VirtualAlloc . . . . .	12
3.4.2	Codice decodificato . . . . .	13
3.5	Secondo VirtualAlloc . . . . .	14
3.6	FUN_0042c820_main . . . . .	16
3.7	FUN_00477050_generate_key_and_files . . . . .	16
3.7.1	FUN_0046e870 . . . . .	19
3.7.2	FUN_0046d530_generate_alphabet . . . . .	20
3.7.3	FUN_0046d340 . . . . .	20
3.7.4	FUN_0046E940 . . . . .	22
3.7.5	FUN_0046db50_decode_bytes . . . . .	23
3.7.6	FUN_0046dac0_read_following_bit . . . . .	23
3.7.7	FUN_0046dcf0 . . . . .	23

3.7.8	FUN_0046d850 . . . . .	27
3.7.9	FUN_0046dc30_read_with_computed_offset . . . . .	28
3.7.10	FUN_0046e340 . . . . .	29
3.7.11	FUN_004021c0_write_param2_on_param1_param3_times . . . . .	29
3.8	FUN_00429ea0 . . . . .	31
4	FUN_00423740_generate_md5_hash_from_stack_charset . . . . .	33
5	FUN_0042d1d0_check_currentProc_win_on_win . . . . .	38
6	FUN_004270f0_search_for_atom . . . . .	39
7	FUN_004211f0_new_struct_appending_string . . . . .	39
8	FUN_00426a40_get_mutex_or_init_events_obj . . . . .	40
9	FUN_0041a6a0_get_event_object_handle . . . . .	41
9.0.1	FUN_00401018_set_EAX_as_SEH_handler . . . . .	43
9.0.2	FUN_0042cf00_adjust_token . . . . .	44
9.0.3	FUN_0041f680_get_path_name_struct . . . . .	44
9.0.4	FUN_0040f4c0_init_UNICODE_struct . . . . .	45
9.0.5	FUN_0040d3a0_fill_malloc_buffer_and_change_SEH_head . . . . .	46
9.0.6	FUN_0040c9d0 . . . . .	47
9.0.7	FUN_0040c3e0 . . . . .	48
9.0.8	FUN_0040be20 . . . . .	48
9.0.9	FUN_00479220_alloca_UNICODE_bytes . . . . .	49
9.0.10	FUN_0040213a_malloc . . . . .	50
9.0.11	FUN_0040b940 . . . . .	50
9.0.12	FUN_0040c840_write_char_in_malloc_buffer . . . . .	50
9.0.13	FUN_00478ee0 . . . . .	51
9.0.14	FUN_0041f560_get_tmp_path_in_struct . . . . .	51
9.1	FUN_00431440_get_half_md5_struct . . . . .	53
9.2	FUN_00418ce0_find_char_in_buffer . . . . .	56
9.3	FUN_00416570_get_CSP . . . . .	58
9.4	FUN_0042cdd0_add_data_to_hash_obj . . . . .	58
9.5	FUN_00430010_get_md5_struct . . . . .	59
9.6	FUN_0042fa00_get_md5_string . . . . .	59
9.7	FUN_0041e310_init_hash_buffer . . . . .	60
9.8	FUN_00417810_init_first_n_byte_of_buffer . . . . .	61
9.9	FUN_0042ccb0_make_md5 . . . . .	62
9.10	FUN_0042cb70_create_md5_hash . . . . .	62

9.11 FUN_0042f110_get_sysroot_and_gen_struct . . . . .	63
9.12 FUN_0042f080_copy_sysroot_in_new_struct . . . . .	64
9.13 FUN_0042ee30_generete_sysroot_struct . . . . .	64
9.14 FUN_0041deb0_fill_sysroot_sruct_buffer . . . . .	65
9.15 FUN_0041cd00 . . . . .	66
9.16 FUN_0041bee0_struct_factory . . . . .	66
9.17 FUN_0041aef0_gen_md5_struct_skeleton . . . . .	68
9.18 FUN_00418080_invoke_malloc . . . . .	68
9.19 FUN_004192f0_reset_struct . . . . .	69
9.20 FUN_0042fdb0_copy_sysroot_struct_and_append_char . . . . .	69
9.21 FUN_00420bf0 . . . . .	70
9.22 FUN_004203a0_copy_param_1_in_this . . . . .	71
9.23 FUN_00401630_copy_sysroot_truct_buffer . . . . .	72
9.24 FUN_0042f430 . . . . .	72
9.25 FUN_0042f2e0 . . . . .	72
9.26 FUN_0041fd90 . . . . .	73
9.27 FUN_0041f9d0 . . . . .	74
9.28 FUN_00419200 . . . . .	74
9.29 __CxxThrowException@8 . . . . .	75
9.29.1 alcune librerie caricate . . . . .	76
<b>10 Punti oscuri</b>	<b>77</b>
<b>11 Note</b>	<b>77</b>
<b>12 to-do list</b>	<b>78</b>
<b>13 Verifica</b>	<b>79</b>

# 1 Traccia dell'homework

## 1.1 Testo

Analizzare il programma eseguibile hw4.ex\_ contenuto nell'archivio hw4.zip (password: "AMW21"). Determinare ogni possibile informazione riguardo alle funzionalita' del programma, riassumendole in un documento che riporti anche la metodologia adottata ed i passi logici deduttivi utilizzati nel lavoro di analisi.

ATTENZIONE: IL MALWARE E' REALE E PUO' PROVOCARE DANNI AI SISTEMI INFORMATICI SE NON OPPORTUNAMENTE CONTROLLATO E MONITORATO.

## 1.2 Scadenza

Tre giorni prima della data d'appello in cui si intende sostenere l'esame orale.

## 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] HW4: `<matricola studente>`"

## 2 Ambiente di lavoro


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

L'ambiente controllato di utilizzo è un sistema operativo Windows 10 virtualizzato con il software *VirtualBox*, in cui sono istallati gli strumenti di monitoraggio.



## 2.0.1 Riepilogo risultati dell'import

### Import Results Summary



Project File Name:	hw4.ex_
Last Modified:	Wed Jan 05 17:15:00 CET 2022
Readonly:	false
Program Name:	hw4.ex_
Language ID:	x86:LE:32:default (2.12)
Compiler ID:	windows
Processor:	x86
Endian:	Little
Address Size:	32
Minimum Address:	00400000
Maximum Address:	004a3fff
# of Bytes:	671744
# of Memory Blocks:	6
# of Instructions:	0
# of Defined Data:	73
# of Functions:	0
# of Symbols:	16
# of Data Types:	33
# of Data Type Categories:	3
Compiler:	visualstudio:unknown
Created With Ghidra Version:	10.0.4
Date Created:	Wed Jan 05 17:15:00 CET 2022
Executable Format:	Portable Executable (PE)
Executable Location:	/home/luca/Scrivania/windowsSharedDir/hw4.ex_
Executable MD5:	97d87414abc48b98a4fffd07c2ddb25a
Executable SHA256:	7b572046df86d9dfad31e6facc4debd2e3d6877a3833bfc1b085903dc3958f2c
FSRL:	file:///home/luca/Scrivania/windowsSharedDir/hw4.ex_?MD5=97d87414abc48b98a4fffd07c2ddb25a
Relocatable:	true
SectionAlignment:	4096

#### Additional Information

```
----- Loading /home/luca/Scrivania/windowsSharedDir/hw4.ex_ -----
[hw4.ex_]: failed to create pointer at 004a310c: Conflicting data exists at address 004a310c to 004a3114
[hw4.ex_]: failed to create pointer at 004a3114: Conflicting data exists at address 004a3114 to 004a3118
[hw4.ex_]: failed to create pointer at 004a3118: Conflicting data exists at address 004a3118 to 004a311c
[hw4.ex_]: failed to create pointer at 004a311c: Conflicting data exists at address 004a311c to 004a311e
[hw4.ex_]: failed to create TerminatedCString at 004a3196: Conflicting data exists at address 004a31a4
[hw4.ex_]: failed to create pointer at 004a3120: Conflicting data exists at address 004a3120 to 004a3124
[hw4.ex_]: failed to create word at 004a31b2: Conflicting data exists at address 004a31a6 to 004a31b2
[hw4.ex_]: failed to create pointer at 004a3128: Conflicting data exists at address 004a3128 to 004a3132
```

OK

## 2.0.2 Informazioni aggiuntive

```
----- Loading /home/luca/Scrivania/windowsSharedDir/hw4.ex_ -----
[hw4.ex_]: failed to create pointer at 004a310c: Conflicting data exists at
address 004a310c to 004a310f
[hw4.ex_]: failed to create pointer at 004a3114: Conflicting data exists at
address 004a3114 to 004a3117
[hw4.ex_]: failed to create pointer at 004a3118: Conflicting data exists at
address 004a3118 to 004a311b
[hw4.ex_]: failed to create pointer at 004a311c: Conflicting data exists at
address 004a311c to 004a311f
[hw4.ex_]: failed to create TerminatedCString at 004a3196: Conflicting data exists
at address 004a31a4 to 004a31a5
[hw4.ex_]: failed to create pointer at 004a3120: Conflicting data exists at
address 004a3120 to 004a3123
[hw4.ex_]: failed to create word at 004a31b2: Conflicting data exists at address
004a31a6 to 004a31b2
[hw4.ex_]: failed to create pointer at 004a3128: Conflicting data exists at
address 004a3128 to 004a312b
[hw4.ex_]: failed to create word at 004a31c2: Conflicting data exists at address
004a31b4 to 004a31c2
[hw4.ex_]: failed to create pointer at 004a3130: Conflicting data exists at
address 004a3130 to 004a3133
[hw4.ex_]: failed to create pointer at 004a3138: Conflicting data exists at
address 004a3138 to 004a313b
[hw4.ex_]: failed to create word at 004a31da: Conflicting data exists at address
004a31ce to 004a31da
Delay imports detected...
Searching for referenced library: USER32.DLL ...
Unable to find external library: USER32.DLL
Searching for referenced library: SHELL32.DLL ...
Unable to find external library: SHELL32.DLL
Searching for referenced library: CMUTIL.DLL ...
Unable to find external library: CMUTIL.DLL
Searching for referenced library: SHIMENG.DLL ...
Unable to find external library: SHIMENG.DLL
Searching for referenced library: KERNEL32.DLL ...
Unable to find external library: KERNEL32.DLL
Finished importing referenced libraries for: hw4.ex_
[CMUTIL.DLL] -> not found
```

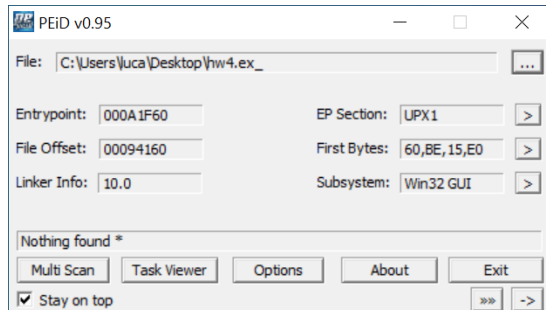


[KERNEL32.DLL] -> not found  
[SHELL32.DLL] -> not found  
[SHIMENG.DLL] -> not found  
[USER32.DLL] -> not found

## 3 Analisi

### 3.1 Basic static (code) analysis

Utilizzando il comando `strings` da terminale *Linux*, si nota come le sembrano essere cifrate. È ragionevole pensare che il programma sia stato impacchettato. Una conferma la troviamo analizzando il file con il tool *PEiD*:

The screenshot shows the 'Section Viewer' window with a table of sections. The table has columns: Name, V. Offset, V. Size, R. Offset, R. Size, and Flags.

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
UPX0	00001000	0000D000	00000200	00000000	E0000080
UPX1	0000E000	00095000	00000200	00094200	E0000040
.rsrc	000A3000	00001000	00094400	00000200	C0000040

Nella sezione EP Section notiamo UPX1: significa che l'eseguibile è stato effettivamente impacchettato con il *packer* open source UPX, che viene però identificato solo dopo aver eseguito una *Deep scan*: UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo. Si può quindi scompattare automaticamente eseguendo `upx -d hw4.ex_ -ounpacked.ex_` sul sistema host. Il file ottenuto è leggermente più grande del file originale: potrebbe aver funzionato: eseguendo nuovamente il comando `strings` sul file `unpacked.ex_` si possono leggere molte stringhe note, come ad esempio "kernel32.dll". Il file ottenuto, però, non risulta eseguibile.

### 3.2 Basic dynamic (behaviour) analysis

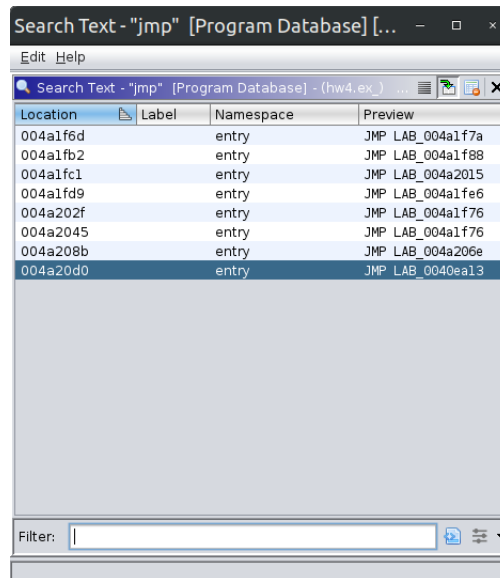
Eseguendo il malware su una macchina virtuale *Windows* non collegata alla rete, si può osservare che questo si tratta in realtà di un *ransomware*: dopo aver terminato la sua esecuzione, vengono creati due nuovi file, un file `asasin.html` e un file `asasin.bmp`, che riportano la stessa scritta; si tratta delle istruzioni da seguire per ottenere la chiave per decifrare i dati. L'immagine viene anche impostata come sfondo. Inoltre, per ogni file che viene criptato dal malware viene creato un file in formato `asasin`.

### 3.3 Advanced analysis

Cerchiamo per prima cosa di recuperare l'eseguibile originale. Come detto, usando *upx* il programma di output non risulta eseguibile. Si procede quindi in altri modi.

### 3.3.1 Ricerca dell'OEP

Utilizzando *Ghidra* per identificare i vari JMP, notiamo che ce n'è qualcuno veramente grande:



In particolare, il salto evidenziato è veramente grande, e potrebbe essere effettivamente il nostro *tail jump*. Tuttavia, per una conferma, sfruttiamo *OllyDB*: il codice, infatti, utilizza nell'*entry point* dello stub l'istruzione **PUSHAD** e si può quindi verificare che i registri verranno recuperati prima del salto evidenziato. Il codice si interrompe proprio prima di quel salto: si può concludere che l'*OEP* sia effettivamente all'indirizzo 0x0040ea13.

### 3.3.2 Fare il dump

Utilizzando il plugin *OllyDump*, si prova a questo punto a eseguire il dump di memoria con tutti e 3 i metodi:

1. ricreando la *IAT* tramite istruzioni di **JMP** e **CALL**
2. ricreando la *IAT* tramite il nome delle *API* invocate
3. senza ricreare la *IAT*

Entrambi i primi due metodi ricostruiscono l'*IAT* correttamente: provando ad invocarli, infatti, si verifica lo stesso comportamento del malware originale. Si procede ad analizzare il file eseguibile generato dal metodo 1.

### 3.4 Advanced analysis - dump-method1.exe

Il programma fa una serie di invocazioni alla funzione `IsBadStringPtrW` che ritorna 0 se non ci sono errori, passando come parametri:

- "lwpqabklbeiutlcti", 17. Valore di ritorno 0.
- "lwpqabklbeiutlcti", 17. Valore di ritorno 0.
- "lwpqabklbeiutlcti", 17. Valore di ritorno 0.
- "vickhgfqkhpdlvlnva", 17. Valore di ritorno 0.

Capire bene pure questa cosa.

#### 3.4.1 Primo VirtualAlloc

All'indirizzo 0040699a si invoca indirettamente la funzione `VirtualAlloc`, allocando 688h bytes, cioè 1672. I parametri di invocazione sono i seguenti:

- `addr = NULL`: i bytes vengono allocati su una qualsiasi zona di memoria disponibile. Significa che ad ogni esecuzione questo potrebbe cambiare.
- `size = 688h = 1672 bytes` allocati.
- `AllocType = MEM_COMMIT`
- `Protect = PAGE_EXECUTE_READWRITE`

Questa memoria viene scritta nelle successive istruzioni, partendo dall'indirizzo 004069ac

```

LAB_004069ac
004069ac ff 31      PUSH    dword ptr [ECX]
004069ae 58             POP     EAX
004069af f8             CLC
004069b0 83 d1 04      ADC     ECX,0x4
004069b3 f7 d0         NOT     EAX
004069b5 f8             CLC
004069b6 83 d8 21      SBB     EAX,0x21
004069b9 8d 40 ff      LEA     EAX,[EAX + -0x1]
004069bc 29 f0         SUB     EAX,ESI
004069be 29 f6         SUB     ESI,ESI
004069c0 29 c6         SUB     ESI,EAX
004069c2 f7 de         NEG     ESI
004069c4 89 03         MOV     dword ptr [EBX],EAX
004069c6 8d 5b 04      LEA     EBX,[EBX + 0x4]
004069c9 8d 7f fc      LEA     EDI,[EDI + -0x4]
004069cc 83 ff 00      CMP     EDI,0x0
004069cf 75 db         JNZ     LAB_004069ac

```

Si sta leggendo il valore in ECX, lo si decodifica e infine lo si scrive nell'area di memoria allocata. Il valore iniziale di ECX è l'indirizzo 00496000, in EBX c'è l'indirizzo restituito dall'invocazione della `VirtualAlloc` e in EDI il valore 688h, cioè il numero di bytes allocati: si tratta di un contatore dei bytes che devono ancora essere scritti. Il codice appena scritto viene poi invocato.

### 3.4.2 Codice decodificato

Si esegue quindi il codice nella zona di memoria appena allocata ma, all'offset 1e viene invocata una funzione che esegue `PUSHAD`: il programma potrebbe essere impacchettato più volte.

All'indirizzo 006e064c vengono caricati gli indirizzi di: `HeapAlloc`, `HeapFree`, `GetTickCount`.

All'offset 65h dell'indirizzo ottenuto dalla prima invocazione di `VirtualAlloc` ce n'è un'altra, sempre di 688h bytes. In questa zona di memoria viene copiato tutti i bytes contenuti nella prima. Quando viene eseguito il `RETN`, il codice continua all'offset 79h del nuovo indirizzo restituito, perché questo era stato precedentemente inserito sulla cima dello *stack*.

Una cosa che è possibile notare è che i bytes a partire dall'indirizzo 004069da, indirizzo in cui si invoca il codice decodificato, vengono modificati proprio da questo blocco di codice. La sostituzione di quei bytes avviene all'offset 113h dell'indirizzo ottenuto dalla prima invocazione di `VirtualAlloc`.

### 3.5 Secondo VirtualAlloc

Il codice inizia con l'invocazione della funzione all'offset 605h. All'offset 8e viene invocata una nuova `VirtualAlloc`, che però alloca molti bytes in più delle precedenti due: ne alloca infatti 83a00h, cioè 539136. All'offset 9f, invece, viene invocata una funzione che esegue `PUSHAD` e `POPAD` rispettivamente all'inizio e alla fine della funzione. Potrebbe essere stato impacchettato ancora un'altra volta.

In seguito viene invocata la funzione `VirtualProtect` con i seguenti parametri:

- `address=00400000`
- `size=400`
- `newProtect=PAGE_READWRITE`
- `pOldProtect=0019FEFC`: è l'indirizzo dove viene memorizzata la precedente protezione. Dopo l'invocazione, contiene il valore 2, che corrisponde a `PAGE_READONLY`.

I 400h bytes successivi all'indirizzo 00400000 vengono sovrascritti con quelli appena inseriti nell'indirizzo ottenuto dalla terza invocazione di `VirtualAlloc`. Al termine dell'operazione, viene ripristinata la protezione sui bytes sovrascritti con una nuova invocazione di `VirtualProtect`, che reimposta la protezione a `PAGE_READONLY`.

Dopo una serie di operazioni che non sono ben chiare, viene invocata nuovamente la funzione `VirtualProtect`, all'offset 10c. Cambia la protezione dei 495616 (79000h) bytes a partire dall'indirizzo 401000, impostandola su `PAGE_READWRITE`: sta per scrivere questi bytes. Per prima cosa, infatti, li imposta tutti a 0, poi scrive al suo interno gli 78600h bytes contenuti dall'offset 400h in poi del terzo indirizzo virtuale. Dopo la scrittura, viene di nuovo cambiata la protezione in `PAGE_EXECUTE_READ` con `VirtualProtect`.

Il codice continua ripetendo lo stesso meccanismo, partendo però sta volta da un offset diverso del terzo indirizzo di virtual alloc: prima si partiva dall'indirizzo che puntava alla stringa `.text`, questa volta la stringa puntata è `.rdata`. Si stanno decodificando le varie sezioni. Si prosegue infatti copiando la sezione `.data`, `.reloc` e `.cdata`.

Il codice continua invocando `virtualProtect` sui 1c4h (452) bytes che partono dall'indirizzo 47a0bc. Viene poi invocata `GetProcAddress` per recuperare l'indirizzo della funzione `OpenMutexA` da `Kernel32.dll`, che viene poi salvato all'indirizzo 47a0bc. Sempre con questo meccanismo, vengono caricare e memorizzate negli indirizzi successivi diverse funzioni; sono in tutto 113. Il codice continua reimpostando la `VirtualProtect` a `PAGE_READONLY`. Dopo aver finito di caricare le `API` da `Kernel32`, inizia a caricare quelle di `Advapi32.dll`. Possiamo quindi concludere che questa funzione effettua il caricamento di tutte le `API` di tutte le librerie.

Il codice continua con `PUSH dword ptr FS:[30]`, all'indirizzo `base_address2 + 0x149`: si sta mettendo sullo stack la struttura `PEB`. Bisogna stare attenti: sappiamo infatti che all'offset 2 di questa struttura c'è il campo `being_debugged`. Si sono quindi inseriti dei breakpoints hardware sul singolo byte, sulla word e sulla dword, in modo da catturare qualsiasi tipo di accesso al byte. Tuttavia questa

struttura è utilizzata per accedere alla struttura dati `PEB_LDR_DATA`, all'offset `0xc`. In particolare, si accede alla lista `InLoadOrderLinks` all'offset `0xc`, ordinata secondo l'ordine di caricamento. Si accede al campo `DLLbase` del primo elemento e lo si confronta con l'*entry point* del programma iniziale: `400000`. Il confronto dà esito positivo (perché?).

All'offset `0x187` viene invocata la `VirtualFree` sui bytes allocati dalla terza `VirtualAlloc`; i parametri infatti sono i seguenti:

- `address = base_address3`
- `size = 83a00`
- `FreeType = MEM_DECOMMIT`

Viene poi invocata una funzione `FUN_00000431` che a runtime non esegue nulla, poi si invoca `FUN_000004f5` che invoca `GetModuleHandle` con parametro `NULL`: si sta recuperando un *handle* al file usato per creare il processo corrente. Successivamente, un'altra invocazione che a runtime non fa altro che eseguire `PUSHAD` e `POPAD`, all'offset `199`. Infine, all'offset `1a0` c'è un'invocazione che l'unica cosa che fa è eseguire una `CALL` all'indirizzo di ritorno, con `POP EAX` e `CALL EAX`. L'indirizzo recuperato dallo stack è `base_address2 + 0x1a5`. In quest'ultima funzione, viene eseguito un `JMP` che salta a `402d8f`: siamo saltati all'interno del nuovo *entry\_point*: i bytes, infatti, sono diversi da quelli originali in quanto sono stati sovrascritti all'offset `113`.

Viene invocata `GetStartupInfoW` e dopo `HeapSetInformation` con i seguenti parametri:

- `HeapHandle = NULL`
- `HeapInformationClass = HeapEnableTerminationOnCorruption`. Enables the terminate-on-corruption feature. If the heap manager detects an error in any heap used by the process, it calls the Windows Error Reporting service and terminates the process. After a process enables this feature, it cannot be disabled.
- `HeapInformation = NULL`
- `HeapInformationLength = 0`

Gli ultimi due parametri sono fissi quando si usa `HeapEnableTerminationOnCorruption` come secondo parametro. All'indirizzo `00402c96` c'è `__heap_init` e altre funzioni di libreria. Tra queste, c'è `GetCommandLineA`, che restituisce in `EAX` il puntatore alla stringa ASCII contenente il path completo dell'eseguibile, compreso di apici: `"C:\Users\luca\Desktop\dump-method1.exe"`. In seguito si invoca `GetEnvironmentStringsA`: sta inizializzando l'ambiente. Alla fine, all'indirizzo `00402d37` viene invocata la funzione `FUN_0042c820`: è l'unica invocata ed è quindi il `main` del processo.

### 3.6 FUN\_0042c820\_main

Questa esegue per prima cosa un `GetModuleHandle` con parametro `NULL`: sta recuperando, di nuovo, un *handle* al file usato per creare il processo corrente.

La funzione invoca successivamente `FUN_00477050_generate_key_and_files`. Quando questa funzione ritorna, si verifica che il valore di ritorno non sia 0: se lo è viene invocato `ExitProcess` con valore -2, altrimenti, invoca `FUN_00429ea0` prima di terminare.

### 3.7 FUN\_00477050\_generate\_key\_and\_files

esegue una serie di `JMP`, `NOP` e altre istruzioni che hanno lo stesso effetto di un `NOP`, come `LEA EAX, dword ptr [EAX]`. L'analisi dinamica e statica è resa quindi molto difficile e frustrante da questo meccanismo.

La funzione apre un ciclo per leggere tutti i bytes dell'eseguibile (da 400000 a 489000) con le seguenti istruzioni, inizializzando `EAX = 400000` e `EDX = 488fe8`:

```
loop_start:
CMP EAX, EDX
JC check

range_end:
<...>

check:
MOV ECX, dword ptr [EAX]
TEST ECX, ECX
JZ increment
MOV ESI, ECX
XOR ESI, 0x88bddd8d
CMP dword ptr [EAX + 0x4], ESI
JNZ increment
XOR ECX, 0xddbca2b2
CMP dword ptr [EAX + 0x8], ECX
JZ pattern_found ; <jmp to other stuff>

increment:
INC EAX
JMP loop_start
```



Si esce dal ciclo con l'istruzione `JZ LAB_004777a2`, quindi solo se:

$$\begin{cases} \text{ECX} \neq 0x0 \\ \text{ECX XOR } 0x88bdd8d = [\text{EAX} + 4] \\ \text{ECX XOR } 0xddbca2b2 = [\text{EAX} + 8] \end{cases}$$

Dove con `ECX` si indicano i 4 bytes puntati da `EAX`, che viene incrementato nel ciclo. Cerchiamo quindi se si esce dal ciclo perché si sono controllati tutti i bytes del *range* oppure perché le tre precedenti condizioni vengono ritrovate. Per farlo si piazza un breakpoint sulla prima istruzione successiva al salto in `pattern_found`, cioè `477393`, e di `range_end`, cioè `4774a7`. Premendo `F9`, l'eseguibile si fermerà alla prima delle due istruzioni raggiunta.

Il primo breakpoint raggiunto è quello in `477393`: viene trovato il pattern quando `EAX = 488000`. Si ha infatti la seguente sequenza di bytes:

```
[ EAX ]      = C8 3E C0 16
[ EAX + 4 ] = 40 85 1D 9B
[ EAX + 8 ] = 15 82 62 A4
```

e si può verificare facilmente che le tre condizioni precedenti sono tutte vere.

Nel blocco di codice seguente, all'indirizzo `477793`, si effettua una invocazione a `VirtualAlloc` con i seguenti parametri:

- `addr = NULL`
- `size = b07h = 2823 bytes` allocati.
- `AllocType = MEM_COMMIT | MEM_RESERVE`
- `Protect = PAGE_READWRITE`

Successivamente, si accede all'offset `30h` di `FS`: `[18]` per prenderne il secondo byte: si sta controllando se si è sotto un debugger. Evitato il controllo, il codice continua scrivendo i bytes allocati con il seguente ciclo:

```

        loop_start:
MOV EAX, dword ptr [EBP + param2]
MOV CL, DL
AND CL, 0x1f
ROL EAX, CL
MOV ECX, dword ptr [EBP + param2]
ROR ECX, 0x3
ADD EAX, ECX
MOV ECX, EDX
ROR ECX, 0xb
ADD ECX, 0x72462828
XOR EAX, ECX
MOV ECX, dword ptr [EBP + local10]
MOV dword ptr [EBP + param2], EAX
LEA EAX, [EDX + EBX * 0x1]
MOV CL, byte ptr [ECX + EAX * 0x1]
XOR CL, byte ptr [EBP + param2]
MOV EBX, dword ptr [EBP, local_c]
INC EDX
MOV byte ptr [EAX], CL
CMP EDX, ESI
JC loop_start

```

Al termine del ciclo, c'è una nuova invocazione di `VirtualAlloc` all'indirizzo 477428:

- `addr = NULL`
- `size = 6494h = 25748 bytes` allocati.
- `AllocType = MEM_COMMIT | MEM_RESERVE`
- `Protect = PAGE_READWRITE`

L'indirizzo di ritorno viene messo sullo stack all'indirizzo `EBP + c`, viene messo in `EDI` l'indirizzo di `VirtualFree` e viene invocata `FUN_0046e870`.

Quando la precedente invocazione ritorna, il codice continua invocando `FUN_0046E940` con i seguenti parametri:

- L'indirizzo restituito dalla `VirtualAlloc` a 477428, l'ultima eseguita
- Il valore `19fedc`

- L'indirizzo restituito dalla `VirtualAlloc` a 477793, la penultima in ordine cronologico

Quando la funzione ritorna, il codice continua invocando una `VirtualFree` all'indirizzo 47767f.

### 3.7.1 FUN\_0046e870

Questa funzione mette dentro ai registri ESI e ECX rispettivamente gli indirizzi 483698 e 483970 prima di invocare la funzione `FUN_0046d530_generate_alphabet`. I valori inseriti nei registri specificano le aree di memoria da scrivere. Quando quella funzione ritorna, ne viene invocata subito un'altra, per due volte: si tratta di `FUN_0046d340`, passando i seguenti parametri:

	Prima invocazione	Seconda invocazione
Primo parametro:	DAT_00483678	DAT_00483658
Secondo parametro:	DAT_00483934	DAT_004838f8
Terzo parametro:	0x3	0x1

Dopo queste invocazione e aver scritto complessivamente i seguenti bytes, la funzione continua sovrascrivendone 2 e ritornando il valore 102h.

FUN_0046d530_ generate_alphabet		
Da	A	Bytes scritti
483970	483984	20
483990	483bd0	576
483698	4836a4	12
4836b8	4836f8	64
FUN_0046d340 Prima invocazione		
Da	A	Bytes scritti
48367c	483696	26
483934	483970	60
FUN_0046d340 Seconda invocazione		
48365a	483676	28
4838f8	483934	60

**Nota:** con le due invocazioni di `FUN_0046d340`, nel suo secondo ciclo, si scrive un'area di memoria unica di 120 bytes che va da 4838f8 a 483676.

### 3.7.2 FUN\_0046d530\_generate\_alphabet

ECX viene fatto puntare a 483970 e i bytes che quell'indirizzo contiene vengono sovrascritti con l'istruzione `STOS dword ptr [EDI]`. Si sta di fatto azzerando i primi 4 bytes, dato che EAX = 0x0.

L'operazione si ripete altre 2 volte, e poi viene eseguita `STOS word ptr [EDI]`: si sono azzerati i primi 14 bytes dall'indirizzo 483970. I successivi 2 bytes vengono modificati in 18: si hanno quindi 3 dword nulle e una che contiene il valore 1800h. I bytes successivi sono cambiati in 98 00 70 00. Si continua a scrivere i bytes a 483990, scrivendo 00 01. Poi ci sono gli incrementi di ECX e EAX in modo da scrivere i successive 48 bytes, cioè 24 iterazioni che scrivono 2 bytes alla volta.

Tabella riassuntiva bytes scritti:

Da	A	Bytes scritti	Note
483970	483984	20	Solo gli ultimi 4 bytes sono diversi da 0 e contengono il valore 00001800
483990	4839c0	48	
4839c0	483ae0	288	
483ae0	483af0	16	
483af0	483bd0	224	
483698	4836a4	12	Solo gli ultimi 4 bytes sono diversi da 0 e contengono il valore 00002000
4836b8	4836f8	64	

Il risultato più evidente di questa funzione è quello di generare nell'area di memoria definita dal range {483990, 483af0} un alfabeto UNICODE.

### 3.7.3 FUN\_0046d340

Questa funzione invoca subito la funzione `FUN_004021c0_write_param2_on_param1_param3_times` con parametri:

	Prima invocazione	Seconda invocazione
Primo parametro:	DAT_00483678	DAT_00483658
Secondo parametro:	0x0	0x0
Terzo parametro:	0x4	0x2

Ma, di fatto, non fa nulla di significativo in entrambe le invocazioni.

La funzione continua scrivendo un certo numero di bytes a partire dall'indirizzo passato come primo parametro con il seguente ciclo e coi successivi parametri iniziali:

```

        loop_start:
MOV EAX, ECX
CDQ
IDIV ESI
MOV byte ptr [EDI + ECX], AL
INC ECX
CMP ECX, EBX
JL loop_start

```

con i seguenti parametri iniziali:

	Prima invocazione	Seconda invocazione
ECX	0x0	0x0
EDX	0x0	0x0
ESI	0x4	0x2
EDI $\equiv$ <Indirizzo base per la scrittura>	0x48367c	0x48365a
EBX $\equiv$ <Numero di bytes scritti>	0x1a = 26 <sub>10</sub>	0x1c = 28 <sub>10</sub>

**Nota:** la seconda invocazione scrive dei bytes in un'area di memoria precedente alla prima e, in particolare, termina la scrittura all'indirizzo 483676, 6 bytes prima dell'indirizzo base usato per la scrittura nella seconda invocazione.

Alcuni di questi bytes verranno poi letti nel ciclo seguente per effettuare un'operazione di decodifica, in modo da poter essere scritti nella zona di memoria passata come secondo parametro.

Il codice continua poi con un altro ciclo, che scrive l'area di memoria passata come parametro 2, usando i bytes contenuti all'indirizzo passato come parametro 1 come fonte per fare operazioni sui byte prima di scriverli, con le seguenti istruzioni:

```

loop_start:
MOV ECX, dword ptr [EBP + param_2]
MOV word ptr [ECX + EAX * 0x2], DX
MOV ECX, dword ptr [EBP + param_1]
MOV CL, byte ptr [EAX + ECX]
XOR ESI, ESI
INC ESI
SHL ESI, CL
ADD EDX, ESI
INC EAX
CMP EAX, 0x1e
JL loop_start

```

Valori iniziali		
	Prima invocazione	Seconda invocazione
EDX	0x3	0x1
EAX	0x0	0x0

I bytes scritti in questo ciclo sono i seguenti:

Prima invocazione		
Da	A	Bytes scritti
483934	483970	60
Seconda invocazione		
4838f8	483934	60

Dopo aver scritto questi bytes, la funzione ritorna.

### 3.7.4 FUN\_0046E940

Per prima cosa, questa funziona alloca sullo stack **4e4h** bytes. In seguito c'è l'invocazione di `FUN_0046dac0_read_following_bit`, che legge il primo bytes presente all'indirizzo restituito da `VirtualAlloc` a 477793, calcola uno `SHR 1` e lo mette sullo stack. Poi semplicemente ritorna il valore 1.

La funzione invoca successivamente `FUN_0046db50_decode_byte`.

Viene invocata la funzione `FUN_0046dcf0`.

Viene invocata la funzione `FUN_0046e340`.

### 3.7.5 FUN\_0046db50\_decode\_bytes

La funzione invoca `FUN_0046dac0_read_following_bit`; l'invocazione è all'interno di un loop e avviene 2 volte. Nel caso la funzione ritorni 1, significa che il byte letto all'indirizzo 800000 diviso per 2 è dispari. In questo caso viene salvato il valore in EDI in EBX: il registro EDI è usato come parametro di controllo per uscire dal ciclo, e logicamente corrisponde alla variabile del ciclo. Non proprio: è così, ma si usa per memorizzare le varie potenze di 2 di cui è composto il byte. Ad esempio, partendo da  $29 = 1dh = 11101_2$ , il ciclo termina quando si raggiunge il valore 20h, cioè il valore 32. Le precedenti potenze di 2 vengono sommate al registro EBX quando il bit relativo è pari 1: come effetto finale, in quel registro verrà ricostruito il valore originale del byte controllato (nel nostro esempio il valore 1dh).

Questo valore viene infine sommato al valore contenuto all'indirizzo passato come secondo parametro alla funzione e il risultato viene restituito.

### FUN\_0046db50\_secondBlock

Per prima cosa si legge in EAX il valore contenuto in ECX; questo valore è l'indirizzo restituito dalla penultima `VirtualAlloc` invocata a 477793, incrementato di 1 (la prima volta): è infatti il blocco di codice che permette di andare avanti con i bytes da controllare: dopo aver controllato il primo byte, passa al secondo in questo blocco. Cosa ci fa con quei byte controllati? qual è lo scopo di dividerli per potenze di 2 e ricostruirlo? I bytes letti e controllati infatti non vengono sovrascritti.

### 3.7.6 FUN\_0046dac0\_read\_following\_bit

La funzione legge il valore sullo stack all'indirizzo contenuto in  $ECX + 8$  ( $0019f9b0 + 8$ ) e lo decrementa, passando da 7 a 6. Poi recupera il byte letto all'indirizzo 800000 e lo divide di nuovo per 2, ritornando poi il risultato di un AND logico tra il vecchio valore del byte e il valore 1. Nella prima invocazione il valore di ritorno è 0, nella seconda è 1: nel secondo caso si ha infatti `AND 3b, 1`, con  $3b = 59$ , dispari.

La funzione viene invocata anche in altri punti, e continua a lavorare allo stesso modo. Il valore di ritorno rappresenta il bit scartato dall'istruzione `SHR`, in modo che, nel caso sia 0, la potenza di 2 corrispondente a quel bit viene aggiunta al registro EBX. Come effetto finale, quando si sono controllati tutti i bit, nel registro EBX ci sarà il valore originale del byte.

### 3.7.7 FUN\_0046dcf0

Questa funzione alloca sullo stack 944 bytes. La funzione invoca la `FUN_0046db50_decode_bytes` per 3 volte con i seguenti parametri:

Prima invocazione		
ECX	5	
struct pointer	address: 19f9b0; values:	
	following byte address	600001
	current byte	1d
	remaining bits	05
offset	101h	
Valori di ritorno		
EAX	11e, salvato in 19f988	
struct pointer	address: 19f9b0; values:	
	following byte address	600001
	current byte	0
	remaining bits	0

Seconda invocazione		
ECX	5	
struct pointer	address: 19f9b0; values:	
	following byte address	600001
	current byte	0
	remaining bits	0
offset	1	
Valori di ritorno		
EAX	1b, salvato in 19f98c	
struct pointer	address: 19f9b0; values:	
	following byte address	600002
	current byte	2
	remaining bits	03



Terza invocazione		
ECX	4	
struct pointer	address: 19f9b0; values:	
	following byte address	600002
	current byte	02
	remaining bits	03
offset	4	
Valori di ritorno		
EAX	e, salvato in 19f990	
struct pointer	address: 19f9b0; values:	
	following byte address	600003
	current byte	3d
	remaining bits	7

Vengono poi azzerati i 17 bytes  $\in [19f844, 19f854]$  prima di entrare in un ciclo in cui si invoca di nuovo la funzione `FUN_0046db50_decode_bytes` con parametri di input ECX e offset rispettivamente pari a 3 e 0. I valori restituiti vengono salvati in un'area di memoria calcolata con  $EBP + ECX - 150h$ , corrispondente a  $19f844 + ECX$ , ECX viene aggiornato ad ogni iterazione con `MOVZX ECX, byte ptr [ESI + 47e5e0]`, con ESI usato come variabile di iterazione del ciclo. Gli indirizzi toccati da questo ciclo sono {19f854, 19f855, 19f856, 19f844, 19f84c, 19f84b, 19f84d, 19f84a, 19f84e, 19f849, 19f84f, 19f848, 19f850, 19f847}:

Address	Hex	dump
0019f844	00	00 00 00 00 00 00 00 00
0019f84c	00	00 00 00 00 00 00 00 00
0019f854	00	00 00 00 00 d8 00 d9 00

Dump prima della scrittura

Address	Hex	dump
0047E5E0	10	11 12 00 08 07 09 06
0047E5E8	0A	05 0B 04 0C 03 00 02
0047E5F0	0E	01 0F 00 00 00 00 00

Offsets da aggiungere a  
19f844 per determinare  
l'indirizzo di scrittura

Address	Hex	dump
0019f844	03	00 00 07 06 04 05 05
0019f84c	04	03 02 02 00 00 00 00
0019f854	05	07 00 00 d8 00 d9 00

Dump dopo la scrittura

La funzione continua invocando `FUN_0046d850`, con i seguenti parametri RIVEDERE:

Parametro 1, ECX	l'indirizzo ECX
Parametro 2	il valore 0x13
Parametro 3	l'indirizzo 19f844
EDX	l'indirizzo 19f5e4

In seguito, recupera il valore 11e da  $EBP - c$  e il valore 1b da  $EBP - 8$ : questi sono i risultati delle prime due invocazioni di `FUN_0046db50_decode_bytes`. Questi valori vengono sommati tra loro e il risultato viene salvato in  $EBP - 10h$ .

Inizia ora un ciclo che inizia caricando in `EAX` l'indirizzo `19f5e4`, che dovrebbe puntare ai byte appena scritti nell'ultima invocazione. Poi invoca la funzione `FUN_0046dc30_read_with_computed_offset`, che prende in input proprio quell'indirizzo insieme a `19f9b0`. Il valore di ritorno viene di fatti confrontato con i valori 16, 17 e 18, ma è previsto anche un ramo `else`. Il confronto, però, avviene attraverso delle sottrazioni:

```
CALL FUN_0046dc30_read_with_computed_offset
MOV ECX, EAX
SUB ECX, 0x10
JZ LAB_46ddd8 ; viene ritornato 16
DEC ECX
JZ LAB_46dc30 ; viene ritornato 17
DEC ECX
JZ LAB_46deec ; viene ritornato 18
; default_case
```

Nel ramo `default_case`, il byte meno significativo del valore ritornato viene messo nell'indirizzo ottenuto da `EBP + ESI - 150h`, che corrisponde a `19f844 + ESI`, con `ESI`. Questo ramo termina incrementando la variabile di ciclo.

Nel caso in cui venga restituito il valore `0x10`, si legge il byte scritto nella precedente iterazione e viene salvato in `EBP - 4` prima di invocare `decode_bytes`. Il risultato viene controllato per verificare che sia diverso da 0, e, in questo caso, si invoca `FUN_004021c0_write_param2_on_param1_param3_times` con questi parametri:

Parametro	Valore a <i>run time</i>	Note
Parametro 1	19f99b	Dove andrebbe scritto il valore dell'iterazione corrente
Parametro 2	6	Valore scritto nell'iterazione precedente
Parametro 3	6	Valore restituito da <code>decode_bytes</code>

Nel caso in cui viene ritornato il valore 17, si cambiano solo i parametri di input per la funzione `FUN_004021c0_write_param2_on_param1_param3_times`: il secondo parametro risulta costante e pari a 0.

Il ciclo termina dopo 139h iterazioni, senza mai esplorare il ramo del caso in cui `FUN_0046dc30_read_with_computed_offset` ritorni 18.

Al termine del ciclo, viene invocata la funzione `FUN_0046d850` per due volte con parametri, rispettivamente:

Parametro	Valore
ECX	FFFFFFF3 $\equiv$ -13 <sub>10</sub>
EDX	1969c4
Parametro 3	19f844
Parametro 4	11e

Parametro	Valore
ECX	FFFFFFF3 $\equiv$ 286 <sub>10</sub>
EDX	19fc24
Parametro 3	19f962
Parametro 4	1b

### 3.7.8 FUN\_0046d850

La funzione azzerava 8 dword a partire dall'indirizzo contenuto in EDX prima di iniziare un ciclo in cui si legge i bytes all'indirizzo passato come parametro a EBP + 8 (quindi il secondo, ma Ghidra lo considera come terzo parametro), con le seguenti istruzioni:

```
MOV dword ptr EAX, [EBX + c]
MOVZX EAX, byte ptr [ECX + EAX]
LEA EAX, [EDX + EAX * 0x2]
INC word ptr [EAX]
INC ECX
```

partendo con ECX = 0. Si sta quindi accedendo ai bytes scritti prima e, dopo averli moltiplicati per due, si usano come offset per accedere a una zona di memoria di 2 bytes che viene incrementata. Tuttavia i primi 2 bytes dell'area puntata da EDX vengono successivamente azzerati. Si inizia poi un altro ciclo, composto dal seguente codice:

```
loop_start:
MOVZX EBX, dword ptr [ECX]
MOV word ptr [EAX + ECX], SI
ADD ESI, EBX
ADD ECX, 0x2
DEC EDI
JNZ loop_start
```

Si parte con:

ECX	19f5e4
EAX	ffffffc8 = -56 <sub>10</sub>
ESI	0
EBX	0
EDI	10h

Quindi si accede alle 16 parole indicizzate da 19f5e4 in poi per leggerne il contenuto, sommarlo ad ESI e scriverlo in EAX - 56<sub>10</sub>. Vengono quindi modificati i 32 bytes ∈ [19f5e4, 19f604]. Venono poi scritti altri byte di un'altra zona di memoria, utilizzando però sempre i byte scritti precedentemente:

```

    loop_start:
MOV EAX, dword ptr [EBP + 19f844]
CMP byte ptr [EAX + ECX], 0x0
JZ label_1
MOVZX ESI, byte ptr [EAX + ECX]
MOVZX ESI, word ptr [EBP + ESI * 2 - 0x20]
MOV word ptr [EDX + ESI * 2 + 0x20]
MOVZX EAX, byte ptr [EAX + ECX]
LEA EAX, [EBP + EAX * 2 - 0x20]
INC word ptr [EAX]
    label_1:
INC ECX
CMP ECX, dword ptr [EBP + c] ; contiene 0x13
JC loop_start

```

ECX	0
EAX	19f844
ESI	c
EBX	19f9b0
EDI	0

Dopo queste operazioni la funzione ritorna con EAX = 19f844.

### 3.7.9 FUN\_0046dc30\_read\_with\_computed\_offset

Questa funzione prende in input due indirizzi: quello della struttura dati e quello da usare come base per leggere il valore di ritorno.

Definisce al suo interno un ciclo in cui viene invocata la funzione [FUN\\_0046dac0\\_read\\_following\\_bit](#) e il suo valore viene salvato in ESI, sommato al precedente contenuto moltiplicato per 2:

```
CALL FUN_0046dac0_read_following_bit
LEA ESI, [EAX + ESI * 0x2]
```

Al valore così ottenuto viene sottratto il contenuto della **word** nella zona di memoria indicizzata da **EDI**, che contiene inizialmente **param\_2 + 2**. Il ciclo termina solo quando il valore così ottenuto è negativo. La somma dei valori delle varie **dwords** viene memorizzato in **EBX**.

Alla fine del ciclo, si somma al valore negativo in **ESI** la somma delle varie **words** lette, contenuto in **EBX**: si ottiene un offset da aggiungere a **param\_2 + 0x20** per leggere il valore da restituire.

### 3.7.10 FUN\_0046e340

La funzione invoca `FUN_<>_read_with_computed_offset` e confronta il valore restituito con il valore **0x100**. Se sono diversi, inizia un ciclo in cui, per prima cosa, si controlla se il risultato di `FUN_<>_read_with_computed_offset` sia maggiore al valore **0x100**.

La funzione inizia un ciclo in cui viene invocata `FUN_0046dac0_read_following_bit`. Poi c'è un ciclo:

```
XOR ESI, ESI
    loop:
MOV ECX, dword ptr [EBP + 8]      ; param 1 = 19f9b0
CALL read_following_bit
ADD EDI, 0x2
LEA ESI, [EAX + ESI * 0x2]
MOVZX EAX, word ptr [EDI]
ADD EBX, EAX
SUB ESI, EAX
JNS loop
```

Parametro	Valore
EAX	19f5e4
ECX	13, cambiato in 19f9b0
EDX	19f5e4
EBX	0
ESI	0
EDI	19f5e4

### 3.7.11 FUN\_004021c0\_write\_param2\_on\_param1\_param3\_times

Riceve in input 3 parametri:

Parametro	Valore
Parametro 1	
Parametro 2	
Parametro 3	

Si confronta il parametro 3 con il parametro 1 e, se sono uguali, la funzione ritorna. Altrimenti si continua controllando il `byte ptr` passato come parametro 2. Se questo non è 0, c'è ancora un altro controllo: si verifica che il terzo parametro sia minore di  $0x80 \equiv 128_{10}$ . Si ha quindi:

```
if (param_3 == param_2){
    return param_1;
}
else{
    if ((byte) param_2 == 0 && param_3 >= 128 && *DAT_00484a54 != 0){
        /* mai esplorato in invocazioni da fun_46dcf0. Neanche la
           terza condizione viene mai raggiunta in realtà */
        ...
        return;
    }
    ...
}
```

Quando la tripla condizione non viene presa, si salva in `EDI` il valore di `ECX`, poi si controlla che il terzo parametro sia minore di 4. Se non lo è, si nega `ECX` e si mette in `AND` con il valore 3. Se il risultato non è 0, si sottrae al parametro 3 il valore ottenuto e si inizia un ciclo *do-while* in cui si scrive nell'area puntata da `EDI` (che contiene il precedente valore di `ECX`) il `byte ptr` passato come parametro 2 per un numero di volte pari al nuovo valore di `ECX`. In seguito si calcola  $EAX \cdot (2^8 + 1) \cdot (2^{16} + 1)$ , e successivamente si salva in `ECX` il risultato di  $\frac{EDX}{4}$ , mentre `EDX` viene messo in `AND` con il valore 3.

```

edi = ecx;
if (param_3 >= 4){
    ecx = (~ ecx) & 3;
    if (ecx != 0){
        param_3 = param_3 - ecx;
        do {
            *edi = (byte) param_2;
            edi++;
            ecx--;
        } while (ecx != 0)
    }
    eax = (byte) param_2 * (2^8 + 1) * (2^16 + 1);
    // se param_2 era 6, eax diventa 06060606
    ecx = param_3;
    param_3 = param_3 & 3;
    ecx = ecx >> 2;
}

```

Nel caso in cui si ottenga come risultato un valore  $n \neq 0$ , si scrivono  $n$  `dword` in `EDI` con il valore di `EAX`. Sostanzialmente, si sta continuando a fare quello che si faceva nel ciclo *do-while*, operando però 4 bytes alla volta: il valore di `EAX` infatti è del tipo 06060606, supponendo che 6 sia il valore del parametro 2. Questo è giustificato dal fatto che, dato un interon, allora  $n \% 4 == n \& 3$ . Con le ultime 3 istruzioni dello *pseudo-codice* precedente, quindi, si è messo il resto della divisione intera tra il terzo parametro e il valore 4 in `EDX`, mentre il risultato è in `ECX`. Essendo che ogni `dword` sono 4 bytes, si azzerano tante `dword` quante contenute in `ECX`. Successivamente, si continua a scrivere in `EDI` un byte alla volta il valore del `byte ptr` del secondo parametro, per tante volte quanto il resto della divisione intera.

La funzione ritorna in seguito il primo parametro.

Riassumendo, questo ramo della funzione non fa altro che scrivere sull'indirizzo ricevuto come primo parametro il byte preso come secondo parametro per tante volte quanto specificato nel terzo parametro.

### 3.8 FUN\_00429ea0

La funzione invoca subito `FUN_00401018_set_EAX_as_SEH_handler`. Dopo vengono allocati 0x1f4 bytes sullo stack e si salvano alcuni registri. C'è poi un confronto tra `DAT_004831f8_key_and_files_virtual_alloc_address` e `EBX`, azzerato precedentemente. Se fossero uguali, la funzione ritornerebbe. Essendo diversi, però, continua invocando `SetErrorMode` con i seguenti parametri:

Parametro	Valore
ErrorMode	8003

**Nota:** Il valore 8003 corrisponde alla combinazione `SEM_FAILCRITICALERRORS | SEM_NOGPFAULTERRORBOX | SE`

Con questa invocazione, si previene la creazione di:

- Critical-error-handler message box: l'eventuale errore viene inviato al processo stesso
- Windows Error Reporting Dialog
- Message Box per fallimento dell'API `OpenFile`

Successivamente viene invocata `SetUnhandledExceptionFilter`, con cui si imposta la *default routine* per la gestione delle eccezioni come la funzione `FUN_41f820`. Segue un'invocazione di `FUN_0042cf00_adjust_token`, che non riceve parametri.

La funzione continua mettendo in `EAX` l'indirizzo che contiene i dati della chiave e dei file da creare, per poi controllare se `byte ptr [EAX + e] == BL`, con `BL == 0`. In `[EAX + e]` è contenuto il valore 01, quindi si continua invocando `GetSystemDefaultLangID`, che ritorna il valore 540410. Dovrebbe ritornare un valore a 16 bit:

SubLanguage ID	Primary Language ID
<i>bits</i> ∈ [10, 15]	<i>bits</i> ∈ [0, 9]

Il valore 0410 del registro `AX` rappresentano il codice `it-IT`, come riportando in [questo pdf scaricabile dalla documentazione microsoft](#).

Viene controllato il valore di `Primary Language ID`, confrontandolo con il valore `0x0019`, che dovrebbe corrispondere alla sigla `ru`. Poiché il confronto non va a buon fine, viene invocata `GetUserDefaultLangID`, per ottenere lo stesso valore precedente: 0410. A questo punto, dopo aver fatto gli stessi controlli sul byte meno significativo, ripete la sequenza di istruzioni: invocazione a `GetUserDefaultLangID`, `AND AX, SI`, con `SI = 0x3ff` e `CMP AX, 0x19`; segue però un `JNZ` che quindi viene preso. Si continua mettendo in `EAX` l'indirizzo che contiene la chiave RSA e i file di testo da creare, e si recupera la `dword` all'offset 8: `MOV EAX, dword ptr [EAX + 0x8]` per confrontarla col valore 0 contenuto in `EBX`. Il successivo `JBE` non viene preso, perché il valore letto è `0xc`. Viene eseguita una moltiplicazione con segno tra `EAX` e `3e8 = 100010`, e il risultato viene salvato in `EAX`. Viene poi messo sullo stack e usato come parametro di una invocazione di `Sleep`: si dorme per 12 secondi.

Si mette poi in `EAX` l'indirizzo ottenuto da `EBP - 0xac = 19fe30` per poter invocare la funzione `FUN_0041f680_get_path_name_struct` con i seguenti parametri:



Parametro	Valore
Parametro 1	19fe30
Parametro 2	0

Quando la funzione ritorna , si carica in EAX l'indirizzo di EBP - 0xd0 = 19fe0c. Questo valore viene messo sullo stack, prima di impostare a 1 il valore 0 inserito sullo stack dall'invocazione di `FUN_00401018_set_EAX_as_SEH_handler`. Viene poi invocata la funzione `FUN_0041f560_get_tmp_path_in_struct`.

Quando la funzione ritorna, si mette in EAX l'indirizzo che contiene il buffer con la chiave e il testo dei files, **scritto nell'altro ramo del main**. Poi si modifica il valore che la funzione `FUN_00401018_set_EAX_as_SEH_handler` aveva messo a 0 e che era stato precedentemente aggiornato a 1, ponendolo uguale a 2.

Ora si controlla il 12° byte del buffer puntato da EAX e lo si confronta con il valore di BL = 00 con `CMP byte ptr [EAX + 0xc], BL`. Essendo uguali, il successivo JZ viene preso. Questo salto evita l'esecuzione di gran parte del codice della funzione, che quindi non viene analizzato e porta l'esecuzione all'interno di un ciclo.

Si continua mettendo in EAX l'indirizzo di EBP - 0xfc = 19fde0, che viene messo sullo stack prima di invocare `FUN_00431440_get_half_md5_struct`. La funzione ritorna quindi una struttura dati `buffer_handler_ASCII`, col buffer contenente la prima metà dell'md5 dei byte 0x1e 0xfb 0x19. Questa struttura viene copiata nella variabile globale `DAT_00481fb4_half_md5_struct` con la funzione `FUN_004203a0_copy_param_1_in_this`, e poi si invoca `FUN_004192f0_reset_struct` sulla struttura ritornata.

Segue una invocazione di `FUN_00426a40_get_mutex_or_init_events_obj` che non riceve parametri.

Si controlla il valore di ritorno e, se questo è 0, si continua invocando `FUN_004270f0_search_for_atom`. Il valore di ritorno viene confrontato con il valore 0 e, se lo è, si confronta il byte all'offset 0xf della zona di memoria `DAT_004831f8_key_and_files_virtual_alloc_address` con il valore 0. Se sono uguali, si salta gran parte della funzione.

Si continua caricando in EAX l'indirizzo 19fe7c, viene messo sullo stack prima di invocare l'immensa funzione `FUN_00423740_generate_md5_hash_from_stack_charset`. La funzione appena invocata restituisce una struttura contenente una stringa di 16 caratteri. Questa verrà usata come metà di una hash: infatti, il codice continua copiando la struttura appena restituita nella variabile globale a `DAT_00481fb4_half_md5_struct`;

La funzione continua mettendo in EAX l'indirizzo dell'area contenente testo del file e la chiave rsa, e mette sullo stack la dword all'offset 0x1043 l'indirizzo dell'offset 0x1047. Si mette in ECX la variabile globale `DAT_00481fd0` che sembra essere una `handler_buffer_ASCII` con i primi 20 bytes a 0 e il campo `buffer_len` impostato a 0xf. Si invoca poi `FUN_0041f9d0`.

## 4 FUN\_00423740\_generate\_md5\_hash\_from\_stack\_charset

La funzione non riceve parametri. Si invoca Si invoca FUN\_00401018\_set\_EAX\_as\_SEH\_handler con EAX che vale LAB\_00424894.

Vengono azzerati i 10 bytes da 19fca8 a 19fcb5, estremi compresi. Poi si invoca FUN\_00420300\_get\_first\_half\_param3\_chars per copiare i primi 6 caratteri della struttura DAT\_00481fb4\_half\_md5\_struct; la struttura ottenuta è all'indirizzo 19fc90. La stringa ottenuta, "f0883f", viene usata come input di \_strtoul, che la interpreta come il valore esadecimale 0xf0883f, e restituisce quindi quel valore.

Si *reset*-ta la struttura con i primi 6 caratteri dell'hash, e si mette in EAX il valore della dword all'offset 0x103f del buffer DAT\_004831f8\_key\_and\_files\_virtual\_alloc\_address: contiene 0x1719. Viene poi invocata la funzione GetUserDefaultUILanguage che restituisce il valore 0x410 che rappresenta il codice it-IT. Questo valore viene poi moltiplicato per  $2^{14}$ , ottenendo 0x1040000; il valore così ottenuto viene messo in XOR con il valore 0x1719, recuperato precedentemente dalla zona di memoria.

Vengono azzerati  $0x98 = 152_{10}$  bytes  $\in [19fbc8, 19fc60)$ . I 4 bytes precedenti vengono invece usati come parametro di GetSystemVersionExA: si tratta di una struttura \_OSVERSIONINFOEXA, perché il primo campo, che ne specifica la dimensione, è impostato a 9c. La struttura dati viene così riempita:

DWORD dwOSVersionInfoSize @ 19fbc4	0x9c
DWORD dwMajorVersion @ 19fbc8	0x06
DWORD dwMinorVersion @ 19fbcc	0x02
DWORD dwBuildNumber @ 19fbd0	0x23f0
DWORD dwPlatformId @ 19fbd4	0x02, che rappresenta VER_PLATFORM_WIN32_NT
CHAR szCSDVersion[128] @ 19fbd8	0. Una stringa che rappresenta l'ultimo <i>service pack</i> installato. Se la stringa è vuota, non è installato alcun <i>service pack</i> .
WORD wServicePackMajor @ 19fc58	0
WORD wServicePackMinor @ 19fc5a	0
WORD wSuiteMask @ 19fc5c	0x0100, corrispondente a VER_SUITE_SINGLEUSERS. Specifica che è supportato <i>Remote Desktop</i> ma solo per una sessione interattiva.
BYTE wProductType @ 19fc5e	0x01 corrisponde a VER_NT_WORKSTATION. Specifica che è installato Windows 8, Windows 7, Windows Vista, Windows XP Professional, Windows XP Home Edition o Windows 2000 Professional.
BYTE wReserved @ 19fc5f	0x00

In base ai campi *\*Version*, la versione di windows sarebbe la 8, tuttavia, la versione usata è la 10.

Una nota della documentazione aiuta:

*“Applications not manifested for Windows 8.1 or Windows 10 will return the Windows 8 OS version value (6.2).”*

Si invoca poi `GetSystemMetrics` con parametro `0x59 = 8910`, che corrisponde a `SM_SERVER2`: si cerca il *build number* se il sistema è Windows Server 2003 R2, altrimenti ritorna 0. La funzione ritorna effettivamente 0, ma il valore non viene controllato.

Si procede controllando i campi *\*Version* per trovare quella giusta. Nel ramo di gestione della versione 6.2, si legge il `wProductType` per controllarne il valore. Se questo campo risulta essere 0, allora si imposta `EAX` a 1, altrimenti lo si mette a 0: `eax = (wProductType == 0);`; a questo valore si somma poi 9, e verrà poi messo in `AND` con `0x1f`. Tuttavia, essendo `EAX` uguale o a 9 o a 10, il risultato dell'AND non può cambiare: entrambi i valori sono esprimibili con 4 bit, e i 4 bit meno

significativi di 0x1f sono tutti a 1. Viene recuperato anche il valore di `wServicePackMajor` in `ESI`, che viene poi messo in AND con 0x7 e moltiplicato per  $2^5$ ; essendo 0, il valore rimane 0. Viene poi calcolato l'OR tra `ESI` e `EAX` (0 Or 9) e questo valore viene moltiplicato per  $2^{0x17} = 2^{23}$  che da come risultato:

$$ESI = \begin{cases} 0x4800000 & \text{se } wProductType == 0 \\ 0x5000000 & \text{se } wProductType == 1 \end{cases}$$

Si calcola poi l'AND tra `EDI` e 0x807fffff, con `EDI` che contiene, ricordiamo, 41719, risultato di:

```
void *tmp = *DAT_004831f8_key_and_files_virtual_alloc_address;
int base = *(tmp + 0x103f); // *8a103f = 0x1719
int lang = GetUserDefaultUILanguage();
edi = lang << 14 + base;
// altre operazioni nel mezzo, ma sono ridondanti per questo run
```

Si continua invocando la funzione `FUN_0042d1d0_check_currentProc_win_on_win`. Questo valore viene moltiplicato per  $2^{31}$ , messo in OR con `ESI` che contiene 4841719, e messo sullo stack alla posizione 19fcb0.

Si procede invocando la funzione `DsRoleGetPrimaryDomainInformation` con questi valori:

LPCWSTR lpServer	Name del computer da cui recuperare le informazioni. Se <code>NULL</code> , allora si usa il computer locale.
DSROLE_PRIMARY_DOMAIN_INFO_LEVEL InfoLevel	Valore che specifica il tipo di dati da recuperare e la struttura di output. Con il valore 1 interpreta la struttura seguente come <code>_DSROLE_PRIMARY_DOMAIN_INFO_BASIC</code>
PBYTE *Buffer [out]	Puntatore all'indirizzo di un buffer.

```
C++
typedef struct _DSROLE_PRIMARY_DOMAIN_INFO_BASIC {
    DSROLE_MACHINE_ROLE MachineRole;
    ULONG                Flags;
    LPWSTR               DomainNameFlat;
    LPWSTR               DomainNameDns;
    LPWSTR               DomainForestName;
    GUID                 DomainGuid;
} DSROLE_PRIMARY_DOMAIN_INFO_BASIC, *PDSROLE_PRIMARY_DOMAIN_INFO_BASIC;
```

*Struttura dati usata*

Si continua recuperando in **EAX** il campo **MachineRole**, che contiene il valore 0. Si fanno vari controlli, ma se è minore di 2 e diverso da 1, si continua facendo un AND tra la parola a **19fcb4** (0x0000) e **0xffff8**.

Si continua poi invocando **DsRoleFreeMemory**, liberando la memoria precedentemente allocata.

Si continua mettendo in **ECX** il valore costante **0x50b** e si inizia poi un ciclo di 10 iterazioni in cui:

```
for (eax = 0; eax < 10; eax++){
    ecx = 0x50b
    dx = cx + (byte) *(19fca0 + eax)
    dx = dx * 0x1e0b
    ecx = dx
}
```

I bytes puntati dalla zona di memoria sono quelli che all'inizio della funzione vengono inizializzati a 0, e che poi vengono modificati in

**[0x3f, 0x88,0xf0, 0x03,0x19,0x17,0x84,0x84,0,0]** .

Il risultato di questo ciclo viene messo in **EAX**, moltiplicato per  $2^3$ .

In un secondo ciclo, si modificano gli 8 bytes a partire dall'indirizzo **19fc7e** con:

```
edi = 0
ecx = 0x883f
eax = 0x5e48
edx = 8
do{
    al = cl
    ecx = (ecx * 0x1f0d) // 32bits size kept
    bl = 0x51
    eax = bl * al
    al = al XOR *(19fcae + edi)
    *(19fc7d + edi) = al
    ecx = cx
    edx--
    edi++
} while(edx != 0)
```

Il codice continua modificando lo stack in vari punti; tra le altre cose, costruisce la stringa "YBNDRFG8EJKMCPQXOT1UWISZA345H769" nei 32 bytes all'indirizzo **19fc8c**, senza però il terminatore di stringa. In più, resetta la struttura dati di tipo **nuffer\_handler\_ASCII** all'indirizzo **19fc60** a mano, impostando il primo byte del buffer e il campo **wrote\_chars** a 0, e il campo **buffer\_len** a **0xf**.

Questa struttura viene poi passata come parametro `this` per l'invocazione di `FUN_0041deb0_fill_sysroot_struct_buffer` per 16 volte, con cui si scrive nel buffer una stringa di 16 caratteri recuperati dall'insieme definito prima. Le modalità di accesso su come i caratteri vengono scelti dall'insieme costituito da "YBNDRFG8EJKMCPQX0T1UWISZA345H769" non è stato approfondito.

La stringa memorizzata nel buffer della struttura alla fine delle iterazioni è: "86RB6EG6BUPIX9AI". Il codice continua poi con uno schema del genere:

```
current_byte = *(buffer_base + eax);
if (current_byte < 0x41 || current_byte > 0x46){
    if (current_byte < 0x30 || current_byte > 0x39){
        <ret_block>
    }
}
eax++;
if (eax >= struct -> wrote_chars){
    <generate_new_struct>
}
```

Nel blocco chiamato `generate_new_struct`, il codice rigenera sullo stack la stringa dalla quale attingere per recuperare 16 caratteri da inserire nel buffer di una nuova struttura.

Nel blocco chiamato `ret_block`, invece, il codice continua copiando la struttura dati costruita in una nuova struttura all'indirizzo 19fe7c e *reset-tando* quella precedente.

La funzione imposta infine FS: [0] ponendolo a 19fed0 prima di ritornare l'indirizzo della struttura appena copiata.

Questa funzione, quindi, genera una struttura dati `buffer_handler_ASCII` con una stringa di 16 caratteri. È presumibilmente una hash md5.

## 5 FUN\_0042d1d0\_check\_currentProc\_win\_on\_win

Costruisce sullo stack le stringhe "kernel32.dll" e "IsWow64Process" e recupera l'handle alla *DLL* con `GetModuleHandle` e recupera l'indirizzo della funzione con `GetProcAddress`. Se tutto va bene viene invocata `GetCurrentProcess` per recuperare l'handle al processo corrente da usare come parametro per l'invocazione di `IsWow64Process`, il cui indirizzo è stato appena recuperato. Il secondo parametro passato è 19fbac, ed è il parametro di output che rappresenta un puntatore che conterrà un booleano con il risultato dell'invocazione.

## 6 FUN\_004270f0\_search\_for\_atom

La funzione non riceve parametri. Si invoca Si invoca FUN\_00401018\_set\_EAX\_as\_SEH\_handler con EAX che vale LAB\_004273a6.

Viene poi costruita una nuova struttura all'indirizzo 19fc84 con la funzione FUN\_00420fc0\_new\_struct\_with\_combined\_string, passando come parametri:

Parametro	Valore
newStructPointer	19fc84
baseString	47d138 = "~~~"
endStringBuffer	DAT_00481fb4_half_md5_struct

Viene poi invocata la funzione FUN\_004211f0\_new\_struct\_appending\_string con parametri:

Parametro	Valore
Parametro 1	19fca0: probabile struct vuota
Parametro 2	19fc84: struttura appena creata che ha nel buffer la concatenazione di DAT_0047d138_~~~ e la metà dell'md5
Parametro 3	DAT_0047d138_~~~

Viene poi invocata GlobalFindAtom con la stringa appena costruita come parametro. Poiché questa stringa non è stata registrata, nella *global atom table*, viene restituito 0. In questo caso il codice continua provando a cercare la stringa nella *local atom table* invocando FindAtom, ma anche questa fallisce. In questo caso, si azzerava EBX, altrimenti si sarebbe impostato a 1. Questo è l'effettivo valore di ritorno ma, prima di terminare, si imposta FS:[0] ponendolo a 19fed0.

## 7 FUN\_004211f0\_new\_struct\_appending\_string

La funzione riceve 3 parametri:

Parametro	Descrizione
Parametro 1	buffer_handler_ASCII di output
Parametro 2	Struttura dati buffer_handler_ASCII
Parametro 3	Puntatore a una stringa

Questa funzione, senza approfondirla, *appende* alla fine della stringa puntata dal buffer della struttura del parametro 2 la stringa puntata dal parametro 3. Il risultato è memorizzato in una nuova struttura *buffer\_handler\_ASCII* che verrà puntata dal parametro 1.

Non c'è differenza con la funzione FUN\_00420fc0\_new\_struct\_with\_combined\_string; ma qui la struttura è passata come secondo parametro invece che come terzo. Inoltre, la struttura dati passata come secondo parametro viene azzerata.

## 8 FUN\_00426a40\_get\_mutex\_or\_init\_events\_obj

La funzione non riceve parametri.

Si invoca `FUN_00401018_set_EAX_as_SEH_handler` con parametro `EAX = LAB_004270ad`. La funzione inizializza a mano un `buffer_handler_ASCII` all'indirizzo `19fc8c`, inizializzando in ordine i campi `buffer_len` a `0x0f`, `wrote_chars` a `0` e `buffer[0]` a `0x0`.

Poi si azzerava `ESI`: lo si userà come variabile di iterazione del ciclo successivo. Questo ciclo, però, viene iniziato solo se il campo `wrote_chars` della struttura dati globale `DAT_00481fb4_half_md5_struct` è maggiore di `0`.

All'interno del ciclo, si leggono i byte dei caratteri che compongono la prima metà dell'hash, vengono incrementati di `1` e messi all'indirizzo `19fcb8` alternandoli col byte `0x61`, che corrisponde al carattere ASCII `a`; alla fine del ciclo, se la prima metà dell'hash fosse `f0883f72d92b0f27`, si otterrebbe la stringa: `Ga1a9a9a4aGa8a3aEa:a3aCa1aGa3a8a` all'interno di una nuova struttura dati, che ha come buffer un puntatore alla stringa, nel campo `wrote_chars` il valore `0x20` e nel campo `buffer_len` il valore `0x2f`.

La funzione continua azzerando `EAX` e costruisce in `EBX - 0x20 = 19fca8` la stringa `Global\`, con terminatore di stringa finale.

Si invoca poi la funzione `FUN_00420fc0_new_struct_with_combined_string` con parametri:

Parametro	Valore	Descrizione
Parametro 1	19fc54	Indirizzo di una <code>buffer_handler_ASCII</code> vuota. È il parametro di output e avrà nel buffer le due stringhe successive combinate
Parametro 2	19fca8	Puntatore alla stringa <code>Global\</code> . appena costruita.
Parametro 3	19fc8c	Indirizzo di una <code>buffer_handler_ASCII</code> che contiene la prima metà dell'hash manipolata.

Questa funzione non è stata approfondita nel dettaglio, ma memorizza nel buffer della struttura ottenuta come primo parametro il risultato della concatenazione tra la stringa passata come primo parametro seguita dalla stringa contenuta nel buffer della struttura passata come secondo parametro. Gli altri campi della struttura di output vengono opportunamente impostati.

Si crea poi anche la stringa `Local\` e la si usa per creare una struttura allo stesso modo della precedente; questa verrà memorizzata in `19fc70`.

La funzione continua invocando `openMutexA`:



Parametro	Valore	Descrizione
dwDesiredAccess	0x100000 $\equiv$ SYNCHRONIZE	.
bInheritHandle	0	Specificando <b>False</b> , non si fa ereditare l'handle di questo mutex ai processi creati
lpName	21f1178	È il puntatore al buffer contenuto nella struttura dati a 19fc54. Si tratta quindi della stringa composta dal prefisso <b>Global\</b> seguita dall'hash manipolata.

Questa funzione restituisce un handle al mutex creato o **NULL** in caso di errore.

Poiché la funzione non ha ancora creato il mutex, la funzione fallisce.

Il valore di ritorno viene salvato in **DAT\_004831f4\_global\_mutex\_handle**, prima di controllare l'effettivo valore.

Nel caso di errore, si prova ad aprire il mutex relativo alla stringa col prefisso **Local\**, eseguendo una nuova invocazione di **openMutexA** con gli stessi altri parametri, tuttavia anche questa invocazione fallisce, ma il valore di ritorno viene comunque salvato in **DAT\_00483200\_local\_mutex\_handle** prima del controllo.

Se anche la seconda invocazione fallisce, si carica in **ESI** la struttura con la stringa con il prefisso **Global\** e si invoca la funzione **FUN\_0041a6a0\_get\_event\_object\_handle**. Il valore di ritorno di questa funzione viene salvato in **DAT\_004831f4\_global\_mutex\_handle**. Una seconda invocazione di **FUN\_0041a6a0\_get\_event\_object\_handle**, con **ESI** che punta però alla struttura il cui buffer inizia con **Local\**, restituisce un handle che viene salvato invece in **DAT\_00483200\_local\_mutex\_handle**. Le tre strutture costruite, le due con le stringhe concatenate nel buffer e quella che contiene l'md5 manipolata, vengono poi *reset-ate* con **FUN\_004192f0\_reset\_struct**.

Viene poi messo in **FS:[0]** l'indirizzo **19fed0** prima di ritornare il valore 0.

## 9 FUN\_0041a6a0\_get\_event\_object\_handle

Azzera **EAX** e **EBX** e invoca la funzione **AllocateAndInitializeSid** con i seguenti parametri:

Parametro	Valore
pIdentifierAuthority	19fc2c
nSubAuthorityCount	1
nSubAuthority0	0
nSubAuthority1	0
nSubAuthority2	0
nSubAuthority3	0
nSubAuthority4	0
nSubAuthority5	0
nSubAuthority6	0
nSubAuthority7	0
*pSid	19fc38 [out] -> 54e9c0

Questa funzione inizializza un *security identifier* (SID), cioè una struttura di lunghezza variabile che identifica *user, group, computer accounts*.

Poi si continua invocando `SetEntriesInAclA`

Parametro	Valore
cCountOfExplicitEntries	1
pListOfExplicitEntries	19fbec
OldAcl	0
*NewAcl	19fc3c, [out] -> 54ec00

*The SetEntriesInAcl function creates a new access control list (ACL) by merging new access control or audit control information into an existing ACL structure.*

Si invoca poi `InitializeSecurityDescriptor` con parametri:

Parametro	Valore
pSecurityDescriptor	19fc0c
dwRevision	1

per inizializzare un nuovo *security descriptor*. Il primo parametro è il parametro di output che punterà a una struttura SECURITY\_DESCRIPTOR.

Si invoca poi `SetSecurityDescriptorDacl` con parametri:

Parametro	Valore
pSecurityDescriptor	19fc0c
bDaclPresent	1
pDacl	54ec00, valore ottenuto precedentemente
bDaclDefaulted	0

aggiungendo la DACL ottenuta precedentemente alla struttura `SECURITY_DESCRIPTOR` passata come primo parametro.

Se nessuna di queste invocazioni fallisce, si procede invocando `CreateEventA` con parametri:

Parametro	Valore
<code>lpEventAttributes</code>	19fc20, puntatore a una struttura <code>SECURITY_ATTRIBUTES</code>
<code>bManualReset</code>	0
<code>bInitialState</code>	0
<code>lpName</code>	21f1178, indirizzo del buffer con la stringa Global

La struttura `SECURITY_ATTRIBUTES` è fatta così:

Parametro	Valore	Descrizione
<code>DWORD nLength</code>	0xc	dimensione della struttura stessa
<code>LPVOID lpSecurityDescriptor</code>	19fc0c	puntatore a una struttura <code>SECURITY_DESCRIPTOR</code> . È quella creata precedentemente
<code>BOOL bInheritHandle</code>	0	specifica se l'handle ritornata è ereditabile dai processi che verranno creati.

Il valore di ritorno viene salvato sullo stack a 19fc34 e viene liberata la struttura `*NewAcl` ottenuta dalla precedente invocazione di `SetEntriesInAclA` invocando `LocalFree`, come specificato nella documentazione. Allo stesso modo, si invoca `FreeSid` sulla zona di memoria puntata da `pSid`, inizializzata nell'invocazione della precedente `AllocateAndInitializeSid`.

La funzione ritorna infine l'handle all'evento creato.

### 9.0.1 FUN\_00401018\_set\_EAX\_as\_SEH\_handler

La funzione invocata aggiunge un nuovo nodo alla `SEH_chain`. Mette sullo stack il valore -1 e il valore contenuto in `EAX`: questo registro viene poi sovrascritto con il primo elemento della catena con l'istruzione `MOV EAX, dword ptr FS:[0]`. Anche questo valore viene messo sullo stack. Ancora, si sovrascrive `EAX` con l'indirizzo di ritorno della funzione: `MOV EAX, dword ptr [ESP + c]`. Ora il nodo viene effettivamente aggiunto alla catena: `MOV dword ptr FS:[0], ESP`, mentre l'indirizzo di ritorno viene modificato: `MOV dword ptr [ESP + c], EBP` e si cambia il valore di `EBP`: `LEA EBP, dword ptr [ESP + c]`. Infine, viene messo sullo stack `EAX`, che contiene il valore di ritorno, prima di eseguire il `RET`.

Il senso di tutto questo è mettere l'indirizzo di ritorno sopra il nuovo nodo: se ciò non venisse fatto, l'istruzione di `RET` cancellerebbe il nodo dallo stack, ma sappiamo che i vari *handlers* sono tutti sullo stack. Inoltre, viene cambiato l'indirizzo di `EBP` facendolo puntare sotto il valore -1, e contiene l'indirizzo del precedente `EBP`.

Lo stack, alla fine, risulta essere:

ESP →	campo prev della SEH chain
	indirizzo del nuovo <i>handler</i> , cioè il valore di EAX avuto in input
	-1
EBP →	precedente valore di EBP

### 9.0.2 FUN\_0042cf00\_adjust\_token

Viene subito invocata `GetCurrentProcess` per recuperare una *pseudo-handler* al processo corrente: si tratta di una speciale costante ( `(HANDLE) -1` ). Si usa questa funzione per motivi di compatibilità con eventuali futuri sistemi operativi. Dopo averla recuperata, viene invocata `OpenProcessToken` con i seguenti parametri:

Parametro	Valore	Note
<code>hProcess</code>	-1	valore restituito da <code>GetCurrentProcess</code>
<code>DesiredAccess</code>	0x80	<code>TOKEN_ADJUST_DEFAULT</code>
<code>phToken</code>	19fcc4	puntatore a <code>handle</code> che identifica il nuovo token aperto

Dopo aver controllato che il valore di ritorno non sia 0, viene invocata `SetTokenInformation` con i seguenti parametri:

Parametro	Valore	Note
<code>hToken</code>	1e8	valore restituito da <code>GetCurrentProcess</code>
<code>infoClass</code>	0x24	<code>TokenVirtualizationEnabled</code>
<code>Data</code>	19fcc0	puntatore a 0 che contiene il valore da impostare
<code>Data_len</code>	4 bytes	dimensione del valore

La funzione termina ritornando il valore restituito da quest'ultima invocazione.

### 9.0.3 FUN\_0041f680\_get\_path\_name\_struct

La funzione riceve 2 parametri:

Parametro	Valore	Note
ECX	19fe30	Primo parametro della funzione invocante
DL	0	
Parametro 1	19f8a8	puntatore all'inizio del buffer che contiene nome dell'eseguibile
Parametro 2	19f8f4	puntatore al terminatore di stringa del buffer che contiene il nome dell'eseguibile

Vengono allocati 418h bytes e si azzerà il valore sullo stack a `[EBP - 4]`. Si mette sullo stack il valore di `ESI`, che contiene `0x3ff`, il valore 208, l'indirizzo di `EBP - 418`, che dovrebbe essere un'area di memoria appena allocata e il secondo parametro della funzione. Si invoca `GetModuleFileName` con i seguenti parametri:

Parametro	Valore
<code>hModule</code>	NULL
<code>PathBuffer</code>	19f8a8
<code>BufSize</code>	208

Si sta recuperando il nome del file eseguibile, salvandolo all'interno di un buffer a 19f8a8 di capacità massima  $0x208=520_{10}$ , mentre viene ritornata la lunghezza della stringa copiata, in numero di caratteri UNICODE.

Viene confrontato il valore ritornato con il valore 208, lunghezza massima del buffer. Si mette in `ECX` il parametro 1 della funzione e si carica in `EAX` l'indirizzo ottenuto con `EBP + EAX * 2 - 418`: si fa puntare `EAX` al primo byte utile dopo il nome dell'eseguibile, sullo stack. Si mette poi questo registro sullo stack, per poi ripristinare il valore di `EAX` precedente, facendolo puntare cioè all'inizio del buffer che contiene il nome dell'eseguibile e anche questo viene messo sullo stack. Si invoca poi la funzione `FUN_0040f4c0_init_UNICODE_struct` con quattro parametri:

Parametro	Valore	Note
Parametro 1	19f8a8	puntatore all'inizio del buffer che contiene nome dell'eseguibile
Parametro 2	19f8f4	puntatore al terminatore di stringa del buffer che contiene il nome dell'eseguibile

Quando la funzione ritorna, si mette in `EAX` il valore del primo parametro, cioè l'indirizzo della struct. Questo era già il valore di `EAX`, che quindi non cambia. Questo valore viene ritornato.

#### 9.0.4 FUN\_0040f4c0\_init\_UNICODE\_struct

La funzione riceve 4 parametri:

Parametro	Valore
<code>ECX</code>	Puntatore a una struttura <code>buffer_handler_UNICODE</code>
<code>DL</code>	???
Parametro 1	Puntatore all'inizio del buffer da inserire
Parametro 2	Puntatore alla fine del buffer da inserire

La funziona inizializza i campi della struttura: imposta a 0 il primo carattere del buffer (quindi i primi 2 bytes, essendo caratteri UNICODE), a 0 il campo `wrote_chars` e a 7 il campo `buffer_len`. Si invoca poi `FUN_0040d3a0_fill_malloc_buffer_and_change_SEH_head`, che non riceve parametri, ma con il seguente stato:

Parametro	Valore
ECX	19fc30
Parametro 1	19f8a8
Parametro 2	19f8f4

Quando la funzione ritorna, si mette in **EAX** il valore di **ESI**, che punta alla struttura dati che è stata allocata, e poi la funzione termina.

### 9.0.5 FUN\_0040d3a0\_fill\_malloc\_buffer\_and\_change\_SEH\_head

Questa funzione sembra non ricevere parametri, ma in realtà utilizza quelli che sono sullo stack dalla precedente invocazione:

Parametro	Valore
ECX	Puntatore a una struttura <b>buffer_handler_UNICODE</b>
Parametro 1	Puntatore all'inizio del buffer da scrivere
Parametro 2	Puntatore alla fine del buffer da scrivere

Si imposta in **EAX** il valore di 0040d560 e poi si invoca **FUN\_00401018\_set\_EAX\_as\_SEH\_handler**. Si mette in **EAX** il valore del secondo parametro a cui si sottrae l'indirizzo del primo: si ottiene il numero di **bytes** scritti nel buffer; questo valore viene poi diviso per due, ottenendo il numero di **caratteri** UNICODE. Si invoca in seguito la funzione **FUN\_0040c9d0** che riceve in input i seguenti 2 parametri:

Parametro	Valore
ECX	19fe30
Parametro 2	numero di caratteri unicode ottenuti: 0x26

La funzione continua azzerando il valore -1 inserito sullo stack dalla funzione **FUN\_00401018\_set\_EAX\_as\_SEH\_handler**, che si trova in **EBP - 4**. Inizia poi un ciclo che itera su tutti i caratteri del path dell'eseguibile, e si esce quando si sono controllati tutti i caratteri. All'interno del ciclo, si legge in **EAX** il carattere UNICODE del path da controllare e lo si mette sullo stack dopo aver messo in **ECX** l'indirizzo della struttura a 19fe30. Una successiva **PUSH** inserisce anche il valore 1, costante, prima di invocare **FUN\_0040c840\_write\_char\_in\_malloc\_buffer**, quindi, coi seguenti parametri:

Parametro	Valore
ECX	19fe30
Parametro 1	1, costante
Parametro 2	il carattere del path attualmente puntato.

Quando la funzione ritorna, viene incrementato il valore di `ESI`, puntatore al path, di 2 bytes: si fa puntare al successivo carattere UNICODE.

Al termine del ciclo, quando tutto il path è stato copiato nell'area di memoria restituita dal malloc, si mette in `ECX` il valore di `EBP - 0xc`, che contiene l'indirizzo del primo nodo della catena SEH (corrisponde infatti a quello contenuto in `FS:[0]`):

next	19fed0
handler	40d560

Per sicurezza, questo nodo viene poi impostato come *head* della catena con `MOV dword ptr FS:[0], ECX` prima di ritornare.

### 9.0.6 FUN\_0040c9d0

La funzione riceve 2 parametri:

Parametro	Valore
this	L'indirizzo della <code>buffer_handler_UNICODE</code>
Parametro 1	Il numero di caratteri UNICODE nel buffer.

Se la funzione non rispetta la seguente condizione, semplicemente termina.

```
struct_ecx.wrote_chars <= param_1 && struct_ecx.buffer_len != param_1
```

Si invoca quindi la funzione `FUN_0040c3e0` con i seguenti parametri:

Parametro	Valore
ECX	19fe30
Parametro 1	numero di caratteri unicode del path: 0x26
Parametro 2	1

La funzione controlla il valore di ritorno e, se è 0, semplicemente ritorna; è il caso in cui sono stati allocati 0 bytes per il path dell'eseguibile. Nel caso sia diverso da 0, come in questo caso, confronta il campo `lunghezza` della struttura dati a `19fe30` con il valore 8. Poi imposta il valore di `[19fe30 + 0x10]` al valore di `EDI`, che contiene 0; anche in quel campo della struttura c'era il valore 0, quindi di fatto non cambia nulla. Poi si controlla il confronto precedente: si verifica che la lunghezza del path sia minore di 8 e, se così non è, si mette in `ESI` l'indirizzo allocato dalla malloc. Si azzerava `EAX` e lo si usa per azzerare la parola a `[ESI + EDI * 0x2]`, ma `EDI` vale 0 e allora si azzerano i primi due bytes, che erano però già stati azzerati. Dopodiché la funzione ritorna il valore 0.

### 9.0.7 FUN\_0040c3e0

La funzione riceve 3 parametri:

Parametro	Valore
ECX	L'indirizzo della <code>buffer_handler_UNICODE</code>
Parametro 1	Il numero di caratteri UNICODE nel buffer
Parametro 2	Il valore 0x1.

Si controlla se la lunghezza del path è minore di 0x7ffffffe e, se non lo è, si passa a un blocco dove si solleva un'eccezione.

Se il valore del campo `buffer_len` è minore del primo parametro si invoca `FUN_0040be20`, apparentemente senza parametri.

Altrimenti c'è un'altra condizione:

```
if (param_2 == 0 || param_1 > 7){
    if (param_1 == 0){
        ...
    }
    this -> buffer[0] = '\0';
}
```

Il blocco più interno non viene mai raggiunto, ma è un blocco che serve a far puntare `this` al buffer nel caso la struttura abbia nei primi 4 bytes del primo campo un puntatore al buffer, piuttosto che il il buffer stesso.

La funzione continua poi azzerando `EAX` e confrontandolo con `ESI`, che contiene il numero di caratteri del path dell'eseguibile, in modo da settare il *carry flag* e sottraendolo a `EAX` tramite `SBB EAX EAX`: si sta impostando `EAX` a -1. Questo valore viene poi negato, ottenendo il valore 1, valore che viene poi ritornato.

**Nota:** se la lunghezza del path fosse stata 0, il carry flag non sarebbe stato impostato e la funzione avrebbe ritornato il valore 0: si sta quindi ritornando il risultato di `param_1 != 0`.

f

### 9.0.8 FUN\_0040be20

**Nota:** questa funzione nel function graph di *Ghidra* non viene mostrata tutta. Per questo motivo, dal punto in cui non viene mostrata, ho creato una nuova funzione in modo da poterla analizzare completamente. La funzione creata è `FUN_0040c146`.



Si invoca `FUN_00401018_set_EAX_as_SEH_handler` dopo aver impostato EAX a 0040c310. Si mette in EDI il numero dei caratteri UNICODE nel buffer (0x26), recuperando questo valore dallo stack e si mette poi in OR con il valore 7, ottenendo 0x27. Si confronta poi il valore ottenuto con 7ffffffe e, se questo è maggiore, si ripristina il valore in EDI al valore originale della lunghezza del path. Altrimenti, come nel nostro caso, si procede azzerando EDX, impostando EBX a 3, e mettendo in EAX il valore ottenuto precedentemente in EDI. C'è poi l'istruzione `DIV EBX`, con cui si divide EAX per il valore di EBX, salvando il quoziente in EAX e il resto in EDX. Nel caso running, si ha

$$\frac{0x27}{3} = 0xd$$

Si continua mettendo in ECX il valore in `[ESI + 14] = 7` per poi memorizzarlo in `EBP - 14`; il valore contenuto in quest'ultima zona di memoria viene diviso per 2 attraverso un `SHR dword ptr [EBP-14], 1`. Questo valore viene poi confrontato con il valore in EAX, che contiene ancora il valore 0xd. Il successivo JBE viene quindi preso.

A questo punto, c'è un `AND dword ptr [EBP-4], 0`: in quella zona dello stack c'è il valore -1, inserito dalla funzione `FUN_00401018_set_EAX_as_SEH_handler`, e lo si sta di fatto azzerando. Si aumenta di 1 la quantità in EDI, cioè il risultato dell'OR iniziale e si salva il risultato in EAX che viene messo sullo stack insieme al valore 0 prima di invocare `FUN_00479220_alloca_UNICODE_bytes`. L'indirizzo allocato restituito viene salvato in `EBP + 8` (param 1?). Si mette in EBX il contenuto di `EBP + c` (param 2?), che contiene il valore 0. Questo valore viene controllato con una istruzione di `TEST`, proprio per verificare se sia 0: il salto viene preso. Si invoca in seguito `FUN_0040b940` con i seguenti parametri:

Parametro	Valore
ECX	19fe30
Parametro 1	1
Parametro 2	0

La funzione continua rimettendo in EAX l'indirizzo allocato dalla malloc precedente, e lo salva all'indirizzo 19fe30, e salvando 0x14 bytes dopo la lunghezza del buffer allocato. Viene poi confrontato il valore della size calcolata (27) con il valore 8 (costante) e il successivo JNC viene preso. Si azzerava ECX e lo si usa per azzerare i primi due bytes all'indirizzo restituito da malloc.

Poi si recupera in ECX il valore del campo `next` dell'ultimo nodo SEH allocato sullo stack e lo si mette in FS:[0]: lo si sta impostando come anello iniziale della catena. La funzione, in seguito, ritorna l'indirizzo del buffer allocato.

### 9.0.9 FUN\_00479220\_alloca\_UNICODE\_bytes

Viene caricato in ECX il valore del parametro ricevuto in input, che di fatto corrisponde a: `(<path_len> OR 7) + 1`, in questo caso a 0x28. Si allocano poi 12 bytes e si azzerava EAX. Si

verifica il valore di ECX e, se è 0, la funzione ritorna. Nel caso analizzato, la funzione continua confrontandolo con 0x7fffffff; il successivo JA non viene preso. Si continua caricando in EAX il valore di  $ECX \times 2$ , cioè  $0x28 * 2 = 0x50 = 80_{10}$ : questo valore corrisponde al numero di bytes di cui è composto il path, incluso il terminatore di stringa. Questo valore lo si mette sullo stack e si invoca `FUN_0040213a_malloc`. Se questa funzione non ritorna 0, come in questo caso, la funzione ritorna l'indirizzo allocato.

#### 9.0.10 FUN\_0040213a\_malloc

La funzione mette sullo stack il valore ricevuto come primo parametro e lo usa per specificare l'argomento della successiva invocazione di `malloc`. Se l'invocazione non fallisce, come in questo caso, la funzione ritorna semplicemente l'indirizzo restituito.

#### 9.0.11 FUN\_0040b940

Si confronta il byte meno significativo del primo parametro con il valore 0. Il successivo JZ non viene preso, essendo questo 01. Tra il confronto e il salto, vengono messi sullo stack i registri ESI, EDI, che vengono sovrascritti rispettivamente con il valore di ECX e il secondo parametro. Nel ramo *else*, viene confrontato il valore di  $[ESI + 0x14] = 7$  con il valore 8; il successivo JC viene quindi preso. Si continua mettendo a 0 i bytes a  $[ESI + 0x10] = [19fe30 + 0x10]$  e a 7 i bytes in  $[ESI + 0x14] = [19fe30 + 0x14]$ . Si modificano poi i 2 bytes in  $[ESI + EDI * 2] = [19fe30 + 0 * 2]$ , impostandoli a 0, prima di ritornare il valore 0.

#### 9.0.12 FUN\_0040c840\_write\_char\_in\_malloc\_buffer

Si mette in ESI il valore di ECX, poi si legge in EAX il contenuto di  $[ESI + 0x10] = [19fe40] = 0$ . Si mette in ECX il valore -1, a cui successivamente viene sottratto il valore di EAX; questo valore viene confrontato con EBX, che contiene il valore del primo parametro (1). Il successivo JA non viene preso. Il campo di quella struttura all'indirizzo 19fe40 sembra essere un valore di controllo: il ramo della funzione saltato, infatti, invoca quella che sembra essere una funzione di errore, perché prima di invocarla mette sullo stack l'indirizzo della stringa `s_string_too_long_0047efe0`.

Si continua controllando il valore di EBX e, se è diverso da 0 come in questo caso, si mette in EDI la somma  $EAX + EBX = 0 + 1 = 1$ , che corrisponde alla somma del campo della struttura dati a 19fe40 e il primo parametro della funzione. Viene poi invocata la funzione `FUN_0040c3e0` con i seguenti parametri:

Parametro	Valore	Descrizione
ECX	19fe30	indirizzo base della struttura dati.
Parametro 1	1	Somma di 19fe40 e il primo parametro della funzione
Parametro 2	0	costante

**Nota:** il parametro 1, nella precedente invocazione, conteneva la lunghezza del path.

Si controlla il valore di ritorno, controllando se sia 0: sappiamo che la funzione invocata ritorna 0 in caso di errore, 1 altrimenti. Se non ci sono errori, si procede invocando la funzione `FUN_00478ee0` con i seguenti parametri:

Parametro	Valore	Descrizione
ECX	19fe30	indirizzo base della struttura dati.
EDX	0	
Parametro 3	0	il campo della struttura a 19fe40
Parametro 4	1	È il primo parametro di FUN_40c840, che è costante
Parametro 5	43	il carattere del path passato come parametro

Quando la funzione ritorna, si procede confrontando il campo lunghezza della struttura dati con il valore 8. Si mette nel campo `caratteri copiati` della struttura il valore di EDI, che contiene il precedente valore della zona incrementato di 1: questo era stato modificato con `LEA EDI, [EAX + EBX]` all'indirizzo 40c90f, con EAX che conteneva il valore precedente del campo della struttura ed EBX che è costante e pari a 1. Si effettua il controllo del precedente confronto con un JC, che non viene preso. Si continua mettendo in EAX l'indirizzo del buffer allocato da malloc e azzerando ECX. Viene poi azzerata la parola successiva a quella appena scritta. Questo meccanismo permette di inserire automaticamente il terminatore di stringa nello stesso momento in cui si scrive l'ultimo carattere del path: nell'invocazione della malloc, infatti, è stato allocato spazio sufficiente per includere anche il terminatore. La funzione, in seguito, ritorna mettendo in EAX il valore di ESI, che punta alla struttura dati in 19fe30.

### 9.0.13 FUN\_00478ee0

Si confronta il quarto parametro con il valore 1. **Se non sono uguali..** Altrimenti, si continua confrontando il valore del campo lunghezza della struttura dati con il valore 8. Il successivo JC non viene preso, e perciò, prima di fare il merge dei rami, mette in ECX il valore contenuto nell'area puntata proprio da ECX: si sta mettendo in questo registro l'indirizzo restituito dalla malloc. Si mette in EAX il parametro 3, e si mette in EDX il valore del quinto parametro, cioè il carattere del path che si sta controllando. Poi si scrive questo carattere nell'area di memoria puntata da `[ECX + EAX * 2]`: si sta copiando il carattere nel punto relativo del buffer allocato con malloc prima di ritornare.

### 9.0.14 FUN\_0041f560\_get\_tmp\_path\_in\_struct

La funzione inizia allocando 0x418 bytes sullo stack. Azzerata la variabile locale EBP - 0x4 e carica in EAX l'indirizzo 19f8ac: in questo indirizzo è memorizzata una stringa molto simile al path dell'eseguibile; è effettivamente la stessa, ma mancano i primi 2 caratteri UNICODE di C:. Questo indirizzo è messo sullo stack. Viene poi messo in ESI il valore 0x208 = 520<sub>10</sub>, e viene poi messo anche lui sullo

stack. Questi valore vengono usati come parametri della successiva invocazione di `GetTempPathW`: il primo parametro specifica la dimensione del buffer passato come secondo parametro, in cui verrà scritto il path di un file temporaneo. Questo path terminerà con un `\`. Questa funzione ritorna la lunghezza del path in `TCHAR`, che è un *typedef* per `char` o `wchar_t`, in funzione del fatto se `UNICODE` è definito o meno. Se il valore di ritorno è maggiore della lunghezza data come primo parametro, allora il valore di ritorno è la lunghezza in `TCHAR` del buffer necessario per memorizzare la lunghezza del path. Se invece la funzione fallisce, ritorna 0.

Si controlla quindi se il valore di ritorno sia zero e, se non lo è, si verifica che il valore di ritorno non sia maggiore di `ESI`, che contiene appunto la lunghezza del buffer usato. Se è andato tutto bene, si mette in `ECX` il valore del primo parametro, e poi si carica in `EAX` il valore di `EBP + EAX * 2 - 0x418`: si stanno riservando sufficienti byte di quelli allocati all'inizio sullo stack per salvare il nome del path temporaneo in caratteri `UNICODE`. Quindi di quei bytes allocati, i primi `len * 2` bytes vengono usati come un buffer, dove `len` è il valore restituito dalla precedente invocazione di libreria. Si fa puntare `EAX` al byte successivo dello stack: se fosse una struttura dati, avrebbe come primo campo un buffer della dimensione restituita e `EAX` punterebbe al secondo campo.

`EAX` viene messo sullo stack, prima di essere sovrascritto ripristinando il valore all'inizio del buffer; anche questo valori viene *pushato*. Abbiamo messo sullo stack il primo e l'ultimo carattere (che è il terminatore di stringa) del path temporaneo restituito. Con questi parametri, viene invocato `FUN_0040f4c0_init_UNICODE_struct`:

Parametro	Valore	Descrizione
<code>ECX</code>	<code>19fe0c</code>	ci sarà una nuova istanza di <code>struct_malloc</code> .
<code>DL</code>	<code>EDX = 530000</code>	non so da dove è uscito, ma contiene molte informazioni.
Parametro 3	<code>19f8ac</code>	inizio del path temporaneo.
Parametro 4	<code>19f8ee</code>	fine del path temporaneo.

La funzione restituisce quindi una struttura dati di tipo `struct_malloc` fatta così:

indirizzo	dimensione in bytes	descrizione
19fe0c	4	indirizzo restituito da malloc, punta a un buffer di dimensione <#DiCaratteriDelPathCompletoDellExe> + 1
19fe10	12	undefined
19fe1c	4	l'offset della word nel buffer allocato da malloc in cui copiare il carattere UNICODE. Rappresenta quindi anche il numero di caratteri copiati.
19fe44	4	l'intero 7, modificato in 27 quando si inserisce l'indirizzo nel buffer al primo campo. È ragionevolmente la lunghezza del buffer in <b>caratteri</b> UNICODE, contando anche il terminatore di stringa.

**Nota:** l'ultimo campo mi aspettavo avesse la stessa dimensione del path temporaneo, invece conserva il valore 0x27.

La funzione restituisce l'indirizzo della struttura.

## 9.1 FUN\_00431440\_get\_half\_md5\_struct

La funzione inizia invocando `FUN_00401018_set_EAX_as_SEH_handler` con EAX che vale LAB\_00431dd4. Alloca poi 0x80 bytes sullo stack. Si mette in EAX l'indirizzo EBP - 0x85 = 19fc38. Si sovrascrive poi l'indirizzo di ritorno della funzione con il valore di ESP; si azzerava EBX e si mette sullo stack il valore di EAX e si modifica il valore di EBP - 0x1c mettendolo a 0, anche se questo valore lo era 0. Poi si invoca `FUN_0042f110_get_sysroot_and_gen_struct`, con un solo parametro:

Parametro	Valore
Parametro 1	Valore di EAX: 19fc38

La funzione continua mettendo sullo stack il valore 0x5c, il valore restituito dalla precedente invocazione (vale a dire l'indirizzo della nuova struct `sysroot_struct`), e il l'indirizzo EBP - 0x54 = 19fc70. Poi mette a 0 il valore -1 inserito dalla precedente `FUN_00401018_set_EAX_as_SEH_handler`, prima di invocare `FUN_0042fdb0_copy_sysroot_struct_and_append_char`:

Parametro	Valore
Parametro 1	19fc70
Parametro 2	19fc38
Parametro 3	5c

Vengono de-allocati 16 bytes sullo stack, poi si azzerà EDI e si salva EBX, che contiene il valore 0, sullo stack. Viene incrementato EDI, mettendolo quindi a 1, prima di salvarlo sullo stack. Si carica in ECX il valore in [EBP - 0x8c] = 19fc38, indirizzo della sysstruct originale che ora ha uno 0 al primo carattere del buffer, prima di invocare `FUN_004192f0_reset_struct`, con questi parametri:

Parametro	Valore
ECX	19fc38
Parametro 1	1
Parametro 2	0

L'invocazione non ha nessun effetto.

Si mette il valore 0x0f in [EBP - 0x24] = 19fca0 e il valore 0x0f in [EBP - 0x28] = 19fc9c. Il byte pointer a [EBP-0x38] = 19fc8c viene messo a 0.

**Nota:** l'indirizzo 19fc8c potrebbe essere un'altra struttura: infatti ha 16 bytes, poi ci sarebbe il campo `next_byte_to_copy` all'indirizzo 19fc9c, impostato a 0 e in seguito il campo `buffer_len` a 19fca0, impostato a 0x0f.

Viene messo sullo stack il valore di EAX, 19fc70, letto nel registro da [EBP-0x54] e che punta alla struttura sysroot copiata. Poi si legge ancora in EAX l'indirizzo 19fc38 da [EBP-0x8c], indirizzo della struttura dati originale e alterata; anche questa viene messa sullo stack. Si mette poi in [EBP - 4], che contiene il valore 0 sotto l'handler SEH, il valore 4 prima di invocare la funzione `FUN_0042f430` con parametri:

Parametro	Valore
Parametro 1	19fc38: struttura dati originale e modificata
Parametro 2	19fc70: struttura dati copiata

La funzione ritorna l'indirizzo della struttura a 19fc38, che punta a una zona di memoria di 176 bytes tutti inizializzati a 0. La funzione continua poi invocando `FUN_004203a0_copy_param_1_in_this`, che mette quell'indirizzo in una nuova struttura dati, all'indirizzo 19fc8c, mentre si ripristina quella a 19fc38 passandola come parametro di `FUN_004192f0_reset_struct`, che non ha di fatto nessun effetto. Si continua invocando `FUN_00418ce0_find_char_in_buffer` coi seguenti parametri:

Parametro	Valore
this	19fc8c
Parametro 1	19fcb0 -> 0x7b
Parametro 2	0
Parametro 3	1

Questa invocazione ritorna -1: il carattere non è stato trovato. La funzione tuttavia non verifica subito il valore di ritorno, ma lo salva in ESI, prima di invocare di nuovo la funzione con questi parametri:

Parametro	Valore
this	19fc8c
Parametro 1	19fcb0 -> 0x7d
Parametro 2	0
Parametro 3	1

Neanche questo carattere viene trovato, e viene quindi ritornato di nuovo il valore -1. Quando anche questa invocazione ritorna, si procede a controllare i risultati delle invocazioni: se una delle due ha restituito -1, si salta un bel pezzo di funzione e si procede invocando `FUN_00416570_get_CSP` con i seguenti parametri:

Parametro	Valore
this	19fca8
Parametro 1	0x18
Parametro 2	0xf0000000 = CRYPT_VERIFYCONTEXT

La funzione ritorna un handle a un CSP. Viene poi invocata la funzione `FUN_0042cb70_create_md5_hash` con parametri:

Parametro	Valore
this	ECX = 19fcac, parametro di output
Parametro 1	19fca8 = CSP ottenuto precedentemente
Parametro 2	0x8003
Parametro 3	0

Si continua confrontando il valore a `[EBP-0x24] = 0xf` con `0x10`, mettendo in EDX `[EBP-0x38] = 19fb1e` e modificando il byte ptr in `[EBP - 0x4]` in `0x9`: è il valore inizializzato da SEH handler.

Si invoca poi la funzione `FUN_0042cdd0_add_data_to_hash_obj` con i seguenti parametri:

Parametro	Valore
this	ECX = 19fcac, puntatore a un handle di hash object
Parametro 1	19fc83
Parametro 2	3

La funzione continua poi caricando in ECX il valore a un certo offset di EBP: `[EBP - 0x8c] = 19fc38`. Questo valore è messo sullo stack, il valore di ECX viene ripristinato a 19fcac e viene invocata `FUN_00430010_get_md5_struct` con parametro 19fc38: questo indirizzo puntava a una `sysroot_struct` contenente il path della `%SYSTEMROOT%`.

La funzione continua invocando poi `FUN_00420300_get_first_half_param3_chars` con parametri:

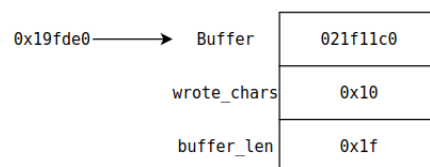
Parametro	Valore
<code>this</code>	19fc38
Parametro 1	19fc54: indirizzo di un <code>buffer_handler_ASCII</code> completamente azzerata
Parametro 2	0
Parametro 3	0x10

Questa funzione non è stata approfondita, ma genera una nuova struttura nell'indirizzo passato come primo parametro il cui buffer contiene i primi `param_3` caratteri della struttura in `this`.

Avendo copiato la hash struct in un'altra struttura, quella precedente (in 19fc38) viene *reset-ata* con la funzione `FUN_004192f0_reset_struct`.

La funzione continua poi verificando che l'hash object usato per criptare non sia `NULL` e, essendo diversi, invoca `CryptDestroyHash`. Allo stesso modo, controlla se l'handle al CSP usato è `NULL` e, se non lo è, invoca `CryptReleaseContext`.

La struttura dati che contiene la prima metà dell'hash viene poi copiata in un'altra struttura, quella all'indirizzo 19fde0 con l'invocazione di `FUN_004203a0_copy_param_1_in_this`. Viene poi resettata la struttura che conteneva la prima metà, quella i cui 3 byte del buffer (19fb1e) sono stati usati per generare l'hash e quella che contiene il path della `%SYSTEMROOT%`. Infine, si imposta `FS:[0]` all'indirizzo 19ff60 e si ritorna l'indirizzo 19fde0:



## 9.2 FUN\_00418ce0\_find\_char\_in\_buffer

Riceve in input 4 parametri:

Parametro	Valore
<code>this</code>	Rappresenta un puntatore a una <code>buffer_handler_ASCII</code> che contiene un indirizzo nel campo buffer: si tratta di una zona di memoria di 178 bytes inizializzati a 0
Parametro 1	Puntatore al byte da cercare nel buffer
Parametro 2	offset per il campo buffer ???
Parametro 3	???

Si verifica il valore del terzo parametro, verificando se sia 0. Prima di eseguire l'istruzione di salto condizionato, si mette in `EAX` il secondo parametro. Dopo aver preso il successivo `JNZ`, la funzio-



ne continua confrontando il valore del secondo parametro in **EAX** con il campo **wrote\_chars** della struttura dati in **ECX**. Questo valore vale 3, perché la struttura dati è fatta così:

Campo	Valore
buffer	19fb1e
wrote_chars	3
buffer_len	0xf

Se il salto non viene preso, si sottrae il parametro 2 (0) al campo **wrote\_chars** (3), e se questo valore ( $3 - 0 = 3$ ) è maggiore del terzo parametro (1) (☒) , allora si imposta **EDX** a  $1 - \text{parametro\_3}$  (0). Il valore ottenuto viene sommato al numero di caratteri scritti, cioè al campo **wrote\_chars** della struttura; il risultato di questa operazione è memorizzato in **EBX**.

Si continua facendo il solito controllo che mette in **ECX** il buffer: si confronta il campo **len** della struttura e, se è maggiore di **0xf**, si recupera il vero buffer interpretando i byte nel campo **buffer** come un puntatore. Tuttavia questo non viene eseguito, nonostante sia abbastanza certo che il **buffer** punti a un ulteriore buffer.

Si somma a **EAX**, che contiene il parametro 2, l'indirizzo del buffer: il secondo parametro è quindi un offset per il campo **buffer**.

Si mette in **ECX** il byte contenuto all'indirizzo passato come parametro 1, estendendolo di segno.

Si mette sullo stack il valore di **EBX**, che contiene **wrote\_chars** +  $(1 - \text{parametro3})$ , e il valore di **ECX**, cioè il byte puntato dal parametro 1 esteso di segno. Si modifica poi il parametro 2, impostandolo al valore di **EAX**, che contiene l'indirizzo del buffer sommato dell'offset del parametro 2. Anche **EAX** viene poi messo sullo stack.

Si invoca poi la funzione `_memchr` con questi parametri:

Parametro	Valore	Descrizione
<code>const void *buffer</code>	19fc8c	Puntatore al buffer della struttura dati sommato dell'offset passato come parametro 2. Rappresenta il buffer in cui cercare il carattere specificato dopo.
<code>int c</code>	$0x7b = 120_{10}$ $= x_{ASCII}$	È il valore puntato dal parametro 1 della funzione esteso di segno. È il carattere da cercare all'interno del buffer.
<code>size_t count</code>	3	È il risultato di <b>wrote_chars</b> + $(1 - \text{parametro3})$ . Rappresenta il numero di caratteri del buffer da controllare durante la ricerca del carattere specificato.

Questa funzione ritorna un puntatore alla prima istanza del carattere nel buffer se ha successo, altrimenti ritorna **NULL** (non viene trovato: ritorna null).

Si controlla il valore di ritorno e, se è **NULL**, allora si imposta **EAX** a -1 e la funzione ritorna.

### 9.3 FUN\_00416570\_get\_CSP

Riceve in input 3 parametri:

Parametro	Valore
this	phProv da usare nell'invocazione di <code>CryptAcquireContextA</code> . È un puntatore a un <i>cryptographic service provider (CSP)</i>
Parametro 1	dwProvType da usare nell'invocazione di <code>CryptAcquireContextA</code>
Parametro 2	dwFlags da usare nell'invocazione di <code>CryptAcquireContextA</code>

La funzione invoca `CryptAcquireContextA` con parametri:

Parametro	Valore
phProv	19fcac, parametro di output
szContainer	0
szProvider	0
dwProvType	0x18, parametro 1 della funzione
dwFlags	CRYPT_VERIFYCONTEXT, parametro 2 della funzione

Il parametro passato come parametro 2 è il valore `CRYPT_VERIFYCONTEXT`: questo flag specifica che non c'è bisogno di accedere a chiavi private. Per questo motivo, il secondo parametro deve essere `NULL`.

L'invocazione ha successo, e viene modificato il parametro di output facendolo puntare a un handle di CSP.

### 9.4 FUN\_0042cdd0\_add\_data\_to\_hash\_obj

La funzione invoca la funzione `CryptHashData` con questi parametri:

Parametro	Valore
hHash	[ECX] = 54e900, handle a hash object
pbData	Parametro 1 = 19fc8c
dwDataLen	Parametro 2 = 3
dwFlags	0

La funzione aggiunge i dati passati come secondo parametro all'oggetto `hash object` passato come primo parametro. Si vogliono criptare i primi 3 bytes puntati da 19fc8c, che contiene 0019fb1e: i bytes criptati saranno 0x1e 0xfb 0x19. L'invocazione ritorna 1, cioè `true`: ha avuto successo. Viene poi ritornato l'indirizzo dei byte inseriti.

## 9.5 FUN\_00430010\_get\_md5\_struct

La funzione riceve un solo parametro:

Parametro	Valore
Parametro 1	indirizzo della <code>sysroot_struct</code> che conterrà il buffer con l'hash.

viene invocata la funzione `FUN_0042fa00_get_md5_string` che non riceve parametri. Questa funzione inizializza una `sysroot_struct` con buffer che punta alla stringa a 32 caratteri dell'hash md5. Il puntatore di questa struttura viene poi ritornato.

## 9.6 FUN\_0042fa00\_get\_md5\_string

Viene invocata la funzione `FUN_00401018_set_EAX_as_SEH_handler` con `EAX` impostato a `LAB_0042fd9f`. Poi si invoca `FUN_0042ccb0_make_md5` con i seguenti 5 parametri:

Parametro	Valore
<code>this</code>	<code>ECX = 19fcac</code>
Parametro 1	2
Parametro 2	19fb64
Parametro 3	19fc00
Parametro 4	0

Viene quindi effettivamente eseguita l'*hash function*, salvando l'output al buffer passato come parametro 2, della dimensione puntata dal parametro 3.

Si invoca poi la funzione `FUN_0041e310_init_hash_buffer` con i seguenti parametri:

Parametro	Valore
<code>this</code>	<code>ECX = 19fbe4</code>
Parametro 1	20
Parametro 2	0

Viene ritornata una struct `sysroot_struct` opportunamente inizializzata. Si confronta il valore di `ESI` (`0x10`) con il valore di `EBX` (`0`) e, se è più grande quello in `ESI`, si inizia un ciclo. In questo ciclo si costruisce la stringa relativa all'hash md5: questa funzione hash, infatti, è rappresentata come una stringa a 32 caratteri, dove ogni carattere è il valore esadecimale che rappresenta mezzo byte.

Nella mia esecuzione, la stringa è: `F0883F72D92B0F270D4D08568153DFC7`, e rappresenta quindi un digest formato dai bytes:

\xf0\x88\x3f\x72\xd9\x2b\x0f\x27\x0d\x4d\x08\x56\x81\x53\xdf\xc7.

Terminato il ciclo, la funzione mette in ESI l'indirizzo 19fc38, che è una `sysroot_struct` che conteneva inizialmente proprio il path di `%SYSTEMROOT%`, i cui campi `next` e `len` vengono impostati rispettivamente a 0 e a 0xf, mentre il primo byte del buffer viene messo a 0; l'indirizzo di questa struttura viene salvato anche in ECX. Si mette in EAX la `sysroot_struct` con il digest md5 e poi questo indirizzo viene messo sullo stack prima di invocare `FUN_004203a0_copy_param_1_in_this`. Si copia quindi la `sysroot_struct` che contiene l'md5 in quella all'indirizzo 19fc38, mentre quella originale viene re-inizializzata.

Si continua invocando la funzione `FUN_004192f0_reset_struct` con parametri:

Parametro	Valore
this	ECX = 19fbe4, indirizzo della struct che precedentemente conteneva l'md5
Parametro 1	1
Parametro 2	0
Parametro 3	1

La precedente invocazione non ha alcun effetto: i campi erano già stati *reset-ati* nella funzione di copia. La funzione imposta poi la base della SEH chain al valore 19fcb8, che è fatto come:

Next	19fed0
Header	d31dd4

Poi la funzione ritorna l'indirizzo della nuova struttura che contiene l'md5.

## 9.7 FUN\_0041e310\_init\_hash\_buffer

Riceve in input 2 parametri:

Parametro	Valore
this	Puntatore a una struttura <code>sysroot_struct</code>
Parametro 1	Il doppio della lunghezza dell'hash md5 ottenuta.
Parametro 2	Viene passato alla seguente invocazione di <code>FUN_00417810_init_first_n_byte_of_buffer</code>

Viene invocata `FUN_0041bee0_struct_factory` con parametri:

Parametro	Valore
this	ECX = 19fbe4
Parametro 1	20
Parametro 2	0, inserito costante

Quando la funzione ritorna si controlla il valore di ritorno e, se il valore non è 0 (è andato tutto bene), invoca `FUN_00417810_init_first_n_byte_of_buffer` con i seguenti parametri:

Parametro	Valore
<code>this</code>	<code>ECX = 19fbe4</code>
Parametro 1	0
Parametro 2	20
Parametro 3	0 = parametro 2 della funzione

Si continua impostando come campo `next` il numero dei bytes scritti dalla precedente invocazione. Poi si mette in `EAX` il puntatore al buffer, facendo il *controllo sulla lunghezza*. Si continua inserendo il terminatore di stringa nel buffer e si ritorna la struttura dati.

## 9.8 FUN\_00417810\_init\_first\_n\_byte\_of\_buffer

Riceve in input 5 parametri:

Parametro	Valore
<code>this</code>	Rappresenta un puntatore a un <code>sysroot_struct</code>
Parametro 1	Valore 0, inserito costante. È l'offset a partire dal quale verranno scritti i bytes
Parametro 2	Il doppio della lunghezza del digest md5. È il numero di bytes che verranno scritti.
Parametro 3	Valore del byte da scrivere nel buffer.

Confronta il parametro 2 con il valore 1. Se i valori sono diversi, confronta il campo `len` della struttura dati con il valore `0x10`. Se questo è più grande, si mette in `ECX` il valore contenuto in `ECX`: si vuole che `ECX` punti a un buffer sufficientemente grande e, evidentemente, la struttura dati prevede che se si è allocato più di 16 bytes, i primi 4 bytes del buffer sono in realtà il puntatore al buffer. Infatti, per come è fatta la struttura, il buffer può memorizzare al massimo  $15 + 1$ . L'informazione su dove si trova effettivamente il buffer è intuibile dalla dimensione della lunghezza nel campo `len`: se questo valore è maggiore di 15, allora il primo campo sarà in realtà un indirizzo che punta al vero buffer.

Si continua mettendo in `EAX` il byte ottenuto come terzo parametro della funzione, estendendone il segno: il valore era 0, quindi si azzerà `EAX`. Poi aggiunge un offset all'indirizzo base del buffer che si trova in `ECX`; l'offset è il parametro 1. Si continua invocando `FUN_004021c0_write_param2_on_param1_param3_times` con parametri:

Parametro	Valore
Parametro 1	<code>021f1178:</code> indirizzo base del buffer allocato con <code>malloc</code>
Parametro 2	0: parametro 3 esteso di segno
Parametro 3	20

La funzione poi semplicemente ritorna l'indirizzo del buffer modificato.

## 9.9 FUN\_0042ccb0\_make\_md5

Riceve in input 5 parametri:

Parametro	Valore
this	Rappresenta un puntatore a un hash object
Parametro 1	Parametro dwParam per la seguente invocazione di CryptGetHashParam
Parametro 2	Parametro *pbData per la seguente invocazione di CryptGetHashParam. È il buffer che conterrà il valore dell'hash.
Parametro 3	Parametro *pdwData per la seguente invocazione di CryptGetHashParam. È il puntatore alla lunghezza del buffer.
Parametro 4	Usato quando la seguente invocazione di CryptGetHashParam fallisce.

Viene invocata CryptGetHashParam, passando come parametri:

Parametro	Valore
hHash	[ECX] = 54e900: l'hash object
dwParam	2 = HP_HASHVAL
*pbData	19fb64
*pdwDataLen	19fc00. Punta al valore 0x80 = 128 <sub>10</sub>
dwFlags	0. Riservato per usi futuri

La dimensione del buffer viene modificata inserendo il numero di bytes scritti, che corrisponde a 0x10: una MD5 è un hash a 128 bit. La funzione controlla l'esito dell'invocazione e se tutto è andato bene ritorna 1.

## 9.10 FUN\_0042cb70\_create\_md5\_hash

La funzione riceve come input quattro parametri:

Parametro	Valore
this	
Parametro 1	puntatore a un handle verso un CSP ottenuto con CryptAcquireContextA
Parametro 2	AlgId da usare nella successiva invocazione di CryptCreateHash
Parametro 3	dwFlag da usare nella successiva invocazione di CryptCreateHash

Azzera il primo campo della struttura dati `this` con un `AND dword ptr [ESI], 0x0`. Mette in `EAX` il valore puntato dal primo parametro: con `MOV EAX, dword ptr [EAX]`; infatti il primo parametro è un puntatore a un handle di CSP. Viene poi invocata la funzione `CryptCreateHash`, che prende in input:

Parametro	Valore
<code>hProv</code>	54ebd0, valore in <code>EAX</code> . Si tratta dell'handle al CSP
<code>AlgId</code>	0x8003, corrispondente a <code>CALG_MD5</code>
<code>hKey</code>	0
<code>dwFlags</code>	0, parametro 3 della funzione
<code>*phHash</code>	19fcac, valore puntato da <code>this</code> . È dove verrà messo l'handle al nuovo <i>hash object</i> .

Si verifica che la funzione non fallisca prima di ritornare il puntatore `*phHash`, parametro di output della precedente invocazione.

## 9.11 FUN\_0042f110\_get\_sysroot\_and\_gen\_struct

La funzione riceve come input un solo parametro:

Parametro	Valore
Parametro 1	Indirizzo in cui generare la struttura dati.

Si allocano 0x108 bytes sullo stack, poi si azzera il valore in `EBP + 8` con una `AND dword ptr [EBP + 0x8], 0`, e si mette in `ESI` il valore 0x108: esattamente la dimensione allocata; questo valore viene poi messo sullo stack. Si carica in `EAX` il valore che punta all'inizio dei byte allocati all'inizio, con `LEA EAX, [EBP - 0x10c]`, per poi metterlo sullo stack. Si invoca quindi `GetWindowsDirectoryA`, con parametri:

Parametro	Valore
Buffer	19fb14
Buffer size	0x104

Viene quindi scritta la stringa `C:\Windows` all'interno del buffer. La funzione ritorna il numero di caratteri scritti se è andato tutto bene, la dimensione che deve avere il buffer se il numero di caratteri da copiare è maggiore della dimensione del buffer passata o il valore 0 in caso di fallimento. Il codice continua quindi controllando se si sono verificati errori e se il buffer è troppo piccolo.

Se tutto è andato bene, si mette sullo stack l'inizio e la fine del buffer allocato sullo stack che contiene la `%SYSTEMROOT%`, per poi invocare la funzione `FUN_0042f080_copy_sysroot_in_new_struct` con i seguenti parametri:

Parametro	Valore
ECX	19fc38
DL	EDX=19fb14
Parametro 3	19fb14
Parametro 4	19fb1e

## 9.12 FUN\_0042f080\_copy\_sysroot\_in\_new\_struct

La funzione prende in input quattro parametri:

Parametro	Valore
Parametro 1	il valore di ECX
Parametro 2	il valore di DL
Parametro 3	Puntatore all'inizio del buffer che contiene la %SYSTEMROOT%
Parametro 4	Puntatore alla fine del buffer che contiene la %SYSTEMROOT%

L'indirizzo in ECX viene salvato in ESI, poi viene messo sullo stack il valore del quarto parametro. Il campo della struttura in ECX in posizione ESI + 0x10 viene messo a 0, anche se lo conteneva già. Il terzo parametro viene messo sullo stack, poi viene impostato a 0xf il valore del campo a ESI + 0x14. Viene inizializzato a 0 il byte iniziale della struttura, poi si invoca `FUN_0042ee30_generete_sysroot_struct`.

La funzione ritorna una struttura fatta nel seguente modo, il cui indirizzo viene restituito.

Struttura in sysroot_struct		
indirizzo	dimensione in bytes	descrizione
19fc38	16	buffer di 16 bytes che contiene il %SYSTEMROOT%
19fe48	4	Prossimo byte da copiare
19fe4c	4	lunghezza massima del buffer

## 9.13 FUN\_0042ee30\_generete\_sysroot\_struct

Riceve 2 parametri (anche se non segnalati da *Ghidra*):

Parametro	Valore
Parametro 1	Puntatore all'inizio del buffer che contiene la %SYSTEMROOT%
Parametro 2	Puntatore alla fine del buffer che contiene la %SYSTEMROOT%



La funzione per prima cosa imposta un altro nodo della catena SEH invocando `FUN_00401018_set_EAX_as_SEH_handler`, con EAX che vale `LAB_0042efd0`. Poi si mette in EAX l'indirizzo di `EBP + 0xc`, che, essendo il secondo parametro, vale `19fb1e`: è il puntatore alla fine del buffer con la `%SYSTEMROOT%`. Si mette sullo stack il valore di EBX, ESI, EDI, poi si modifica il valore di ESI impostandolo a `EBP + 0x8 = 19fc38`, che coincide al primo parametro. Si sottra poi a EAX il valore di ESI: si mette in EAX la lunghezza della `%SYSTEMROOT%`; questo valore è messo sullo stack dopo aver cambiato il valore di `EBP - 0x10` con il valore di ESP. Si modifica poi il valore di `EBP - 0x14`, impostandolo a ECX (che già lo conteneva) prima di invocare `FUN_0041cd00` con i parametri:

Parametro	Valore
ECX	19fc38
Parametro 1	0xa

La funzione continua mettendo a 0 il valore in `EBP - 0x4`, che conteneva il valore -1 inserito da `FUN_00401018_set_EAX_as_SEH_handler`, prima di iniziare un ciclo che termina quando il valore in ESI sarà uguale al valore in `EBP + 0xc`: questi contengono rispettivamente il puntatore all'inizio e alla fine del `%SYSTEMROOT%`.

All'interno del ciclo, si legge il carattere puntato da ESI in EAX, che viene messo sullo stack prima del valore 1, in modo da passarli come parametro alla funzione `FUN_0041deb0_fill_sysroot_struct_buffer`.

Quando questa ritorna, viene incrementato ESI e il ciclo riparte.

La funzione continua ripristinando ECX all'indirizzo sullo stack del nodo della SEH chain inserito all'inizio, per poi impostarlo come valore di `FS:[0]`. La funzione, in seguito, ritorna la struttura `sysroot_struct`.

## 9.14 FUN\_0041deb0\_fill\_sysroot\_struct\_buffer

La funzione prende in input tre parametri:

Parametro	Valore
this	il valore di ECX
Parametro 1	Numero di byte che verranno copiati. 1 - costante
Parametro 2	Carattere da copiare

La funzione, senza analizzarla nel dettaglio, invoca al suo interno `FUN_0041bee0_struct_factory` per copiare il carattere ricevuto in input nella posizione specificata dal campo a `ECX + 0x10`. Ricorda molto il comportamento di `FUN_0040d3a0_fill_malloc_buffer_and_change_SEH_head`.

## 9.15 FUN\_0041cd00

La funzione prende in input due parametri:

Parametro	Valore
this	il valore di ECX
Parametro 1	Lunghezza della %SYSTEMROOT%

Si mette in EAX il valore del primo parametro, e poi si mette in ESI il valore di ECX. Si mette in EDI il valore del campo della struttura a ESI + 0x10, inizializzato precedentemente a 0. Questo valore viene poi confrontato con:

```
CMP EDI, EAX      ;edi = 0; eax = a
JA LAB_0041cdcb
```

Il salto non viene dunque preso. Si esegue poi un altro confronto:

```
CMP dword ptr [ESI + 0x14], EAX    ; esi + 0x14 = f, eax = a
JZ LAB_0041cdcb
```

quindi neanche questo salto viene eseguito. Si procede mettendo sullo stack il valore 1 e quello di EAX, prima di invocare `FUN_0041bee0_struct_factory`.

Quando la funzione ritorna, si controlla il valore di ritorno e, nel caso non sia 0, esegue le seguenti operazioni aggiuntive. Per prima cosa confronta il campo ESI + 0x14 con il valore 0x10:

```
CMP dword ptr [ESI + 0x14], 0x10
<other stuff>
JC LAB_0041cd3c
```

Nel mezzo, si imposta il valore ESI + 0x10 al valore di EDI, che contiene 0 (nessun effetto). Il salto viene preso e si imposta a 0 il valore del byte puntato da EDI + ESI: sono le stesse cose che fa nella funzione `FUN_0041bee0_struct_factory`.

La funzione ritorna il valore ritornato da `FUN_0041bee0_struct_factory`.

## 9.16 FUN\_0041bee0\_struct\_factory

La funzione prende in input tre parametri:

Parametro	Valore
this	il valore di ECX
Parametro 1	Lunghezza della %SYSTEMROOT%
Parametro 2	undefined - riceve 1, costante

Si confronta il valore del parametro 1 con 4294967294. Nel caso fosse maggiore, viene generata una eccezione invocando `FUN_00408066` con parametro `s_string_too_long_0047efe0`.

La funzione continua:

```
if (ecx.max_len >= param_1){
    if (param_2 == 0 or ecx.max_len >= 0x10){
        if (param_1 == 0){
            (1)
        }
    }
    else{
        (2)
    }
}
else{
    (3)
}
```

1.

2. si imposta EAX al massimo tra il secondo parametro e il valore contenuto in `ECX.next_byte`. Si procede invocando `FUN_004192f0_reset_struct` con questi parametri:

Parametro	Valore
this	ECX = 19fc38
Parametro 1	1, costante
Parametro 2	0, max(ecx.next_byte, param_2)

3. Si mette sullo stack il campo `next byte` e la lunghezza passata come parametro 1 e si invoca `FUN_0041aef0_gen_md5_struct_skeleton`.

Si continua azzerando EAX, e confrontando questo valore con il parametro 1:

```
XOR EAX, EAX
CMP EAX, ESI
SBB EAX, EAX
```

ma non c'è nessuna istruzione di jump condizionato successivo. La funzione continua infatti con un `SBB`, che somma a EAX il valore del *carry flag* e sottrae il risultato da EAX stesso; si sta praticamente impostando EAX al valore del carry flag cambiato di segno:

$$eax = eax - (eax + carry) = -carry$$

Una successiva operazione di `NEG EAX` riporta il valore positivo, valore che viene effettivamente ritornato.

## 9.17 FUN\_0041aef0\_gen\_md5\_struct\_skeleton

La funzione sembra non ricevere parametri, invece ne prende tre:

Parametro	Valore
this	il valore di ECX che contiene una <code>sysroot_struct</code>
Parametro 1	la lunghezza di una stringa (20: $2 * \text{len}(\text{md5})$ )
Parametro 2	campo next della struttura dati

Si fa un `OR` tra il parametro 1 e il valore costante `0xf` e il risultato viene diviso per il valore 3 costante con una istruzione di `DIV`: il quoziente sarà in `EAX` e il resto in `EDX`. Il valore del resto viene sovrascritto con la metà del campo `len` della struttura, ottenuto con `SHR` di una posizione. Questo valore viene poi confrontato con `EAX`, che contiene ancora il quoziente della divisione intera. Poiché il valore in `EDX` (7) è minore di `EAX` (15), il successivo `JBE` viene preso. Si carica in `EAX` il valore risultante dall'`OR` precedente (`0x2f`), incrementato di 1 (`0x30`). Questo valore viene messo sullo stack insieme al valore 0 prima di invocare `FUN_00418080_invoke_malloc`:

Parametro	Valore
Parametro 1	0x30

Il valore di ritorno è messo sullo stack, nella posizione occupata dal primo parametro: `EBP + 0x8`. Si confronta il valore di `EBP - 0x14` con il valore 0 e, se sono uguali, si saltano molte istruzioni che dovrebbero operare una copia della `sysroot_struct`. Si invoca poi `FUN_004192f0_reset_struct` con parametri:

Parametro	Valore
this	19fbe4
Parametro 1	1
Parametro 2	0

Questa invocazione non ha però nessun effetto. Si continua salvando nei primi 4 byte del buffer della struttura dati l'indirizzo ritornato da `malloc` e nel campo `len` il valore `0x2f`: è la lunghezza allocata diminuita di 1; si mette poi a 0 il campo `next`.

Si mette in `ECX` il valore in `[EBP - 0xc] = 19fc04`, si azzerava il primo byte dell'indirizzo restituito dalla `malloc` e si mette in `FS:[0]` il valore di `ECX`: si tratta del nodo della SEH chain allocato all'inizio dalla funzione precedente(?). Poi la funzione termina, ritornando l'indirizzo allocato con la `malloc`.

## 9.18 FUN\_00418080\_invoke\_malloc

La funzione riceve in input un solo parametro, e rappresenta il numero di bytes da allocare.

Viene azzerato `EAX` e si controlla se il parametro ricevuto sia minore o uguale a 0; in quel caso si esce subito dalla funzione. Viene poi controllato che il valore non sia più lungo del massimo valore intero:

in quel caso viene sollevata un'eccezione. Se tutto va bene, la funzione continua allocando `param_1` bytes con la funzione `FUN_0040213a_malloc`. Si verifica che questa funzione non abbia errori, e si restituisce l'indirizzo allocato.

## 9.19 FUN\_004192f0\_reset\_struct

La funzione prende in input tre parametri:

Parametro	Valore
<code>this</code>	il valore di <code>ECX</code>
Parametro 1	Valore 1, costante - <i>undefined</i>
Parametro 2	minimo valore tra la lunghezza di <code>%SYSTEMROOT%</code> e il campo <code>ECX + 0x10</code>

Si confronta il parametro 1 con il valore 0:

```
CMP byte ptr [EBP + 0x8], 0x0    ; cmp 1, 0
<other_stuff>
JZ LAB_00419324
```

Nel mezzo, si salva in `EDI` il parametro 2 e in `ESI` il valore di `ECX`. Il salto non viene preso, e si continua confrontando il campo `ESI + 0x14` e il valore `0x10`:

```
CMP dword ptr [ESI + 0x14], 0x10    ; cmp 0xf=15, 0x10=16
JC LAB_00419324
```

Il salto viene preso, e si continua mettendo in `ESI + 0x10` il valore di `EDI`, che contiene 0 (nessun effetto); si ripristina in `ESI + 0x14` il valore `0xf` (nessun effetto); e si mette a 0 il byte puntato da `EDI + ESI`: si usa `EDI` come offset.

Riassumendo, la funzione azzerava il primo byte del buffer, azzerava il campo `next` e imposta a `0xf` il campo `len` della struttura passata in `ECX`. C'è ancora un ramo da esplorare.

## 9.20 FUN\_0042fdb0\_copy\_sysroot\_struct\_and\_append\_char

La funzione riceve 3 parametri:

Parametro	Valore
Parametro 1	Indirizzo <code>19fc70</code>
Parametro 2	Indirizzo della <code>sysroot_struct</code> allocata precedentemente
Parametro 3	valore <code>0x5c</code>

La funzione azzerà il precedente valore di EBP messo sullo stack. Poi invoca la funzione `FUN_0041deb0_fill_sysroot_struct_buffer` con i seguenti parametri:

Parametro	Valore
ECX	19fc38
Numero di bytes da copiare	1
Carattere da copiare	5c

Poiché il campo `next_byte` della struttura vale `0xa`, si inserirà in quella posizione del buffer il carattere `0x5c = \`. Quando la funzione ritorna, se ne invoca subito un'altra: `FUN_00420bf0` con parametro:

Parametro	Valore
ECX	19fc70
Parametro 1	19fc38

Viene ritornato il valore a `[EBP + 8]`, che è il puntatore alla funzione copiata: è lo stesso valore restituito dalla funzione precedentemente invocata.

## 9.21 FUN\_00420bf0

La funzione riceve 2 parametri:

Parametro	Valore
this	Indirizzo in ECX: 19fc70
Parametro 1	Indirizzo della <code>sysroot_struct</code> allocata precedentemente

Viene salvato in ESI il valore di ECX, poi viene azzerato il campo `ESI + 0x10`, si mette il valore `0xf` in `ESI + 0x14` e poi si azzerà il primo byte puntato da ESI sembra si stia inizializzando un'altra `sysroot_struct`. Si invoca poi `FUN_004203a0_copy_param_1_in_this` con i seguenti parametri:

Parametro	Valore
ECX	19fc70
Parametro 1	19fc38

Viene poi restituito il puntatore alla struttura copiata, puntatore che era restituito anche dall'invocazione precedente.

## 9.22 FUN\_004203a0\_copy\_param\_1\_in\_this

La funzione riceve 2 parametri:

Parametro	Valore
this	Indirizzo in ECX: 19fc70
Parametro 1	Indirizzo della sysroot_struct allocata precedentemente: 19fc38

La funzione mette il primo parametro in ESI e il valore di ECX in EDI. Poi confronta EDI ed ESI. Se questi non sono uguali, invoca `FUN_004192f0_reset_struct` con parametri:

Parametro	Valore
this = ECX	19fc70
Parametro 1	1
Parametro 2	0

Poi si confronta il campo `len` della struttura a 19fc38 con il valore 0x10:

```
CMP dword ptr [ESI + 0x14], 0x10
JNC LAB_004203ac
```

Il salto non viene preso, e si continua mettendo in EAX il campo `next_byte` della stessa struttura, che contiene il valore 0xb. Questo valore viene incrementato di 1, e viene messo sullo stack insieme agli indirizzi delle due strutture. Viene quindi invocata `FUN_00401630_copy_sysroot_struct_buffer` con parametri:

Parametro	Valore
Parametro 1	19fc79
Parametro 2	19fc38
Parametro 3	0xc

La funzione continua deallocando 12 bytes sullo stack. Poi si copiano i campi `next_byte` e `len` nella struttura di copia, campi che non erano stati copiati dalla precedente invocazione. La struttura originale viene invece modificata: viene azzerato il campo `next_bytes` e il primo byte del buffer viene annullato.

Viene poi restituito il puntatore alla struttura copiata, che mantiene le informazioni integre.

Riassumendo, la funzione prende una `sysroot_struct` come parametro che viene copiata nella struttura dati in ECX (this).

## 9.23 FUN\_00401630\_copy\_sysroot\_struct\_buffer

La funzione riceve 3 parametri:

Parametro	Valore
Parametro 1	Indirizzo di una <code>sysroot_struct</code> non ancora allocata: 19fc70
Parametro 2	Indirizzo di una <code>sysroot_struct</code> allocata precedentemente: 19fc38
Parametro 3	Bytes da copiare

Senza approfondirla molto, la funzione copia nella `sysroot_struct` in parametro 1 la `sysroot_struct` in parametro 2. L'una cosa che viene copiata è il buffer: infatti il campo `next_byte` della struttura nel primo parametro rimane settato a 0.

RIVEDERE L'USO DEL PARAMETRO 3

## 9.24 FUN\_0042f430

La funzione riceve due parametri:

Parametro	Valore
Parametro 1	Indirizzo della <code>sysroot_struct</code> originale e modificata: presenta come primo carattere del buffer un 00
Parametro 2	Indirizzo della <code>sysroot_struct</code> copiata, con le informazioni integre.

La funzione non fa niente di significativo all'inizio, tranne controllare il campo lunghezza della funzione ricevuta come secondo parametro e confrontarla con 0x10. Dopodiché viene invocata `FUN_0042f2e0` con gli stessi parametri di questa funzione.

## 9.25 FUN\_0042f2e0

La funzione riceve due parametri:

Parametro	Valore
Parametro 1	Indirizzo della <code>sysroot_struct</code> originale e modificata: presenta come primo carattere del buffer un 00
Parametro 2	Indirizzo della <code>sysroot_struct</code> copiata, con le informazioni integre.

Vengono allocati  $0x10c = 268_{10}$  bytes sullo stack, poi si esegue una `PUSH 0x104`: inserisco sullo stack il numero di bytes allocati meno 8. Potrebbe aver allocato una struttura dati che include un buffer di size 0x10c bytes e successivamente inserito la lunghezza del buffer. Il codice recupera l'indirizzo base di questa ipotetica struttura, con `LEA EAX, dword ptr [EBP - 0x10c]`; questo



indirizzo viene messo sullo stack, seguito dall'indirizzo ricevuto in input come secondo parametro. Poiché successivamente viene invocata la funzione `GetVolumeNameForVolumeMountPointA`, quest'ultimo valore messo sullo stack è il primo campo della struttura, cioè il path della `%SYSTEMROOT%`. I parametri passati sono quindi:

Parametro	Valore
Volume mount point	System root path
VolumeName	19fafc
BufferLength	104

La funzione restituisce 0: c'è stato un errore. Il `LastErr` è `ERROR_NOT_A_REPARSE_POINT`. Questo errore viene recuperato invocando `GetLastError`: viene restituito il codice 1126 in `EAX`, che viene poi salvato in `[EBP-0x4]`. C'è poi una `PUSH 0047ee48` e viene letto in `EAX` il valore in `[EBP - 0x8]` = 19fc00 e viene messo sullo stack. Si mette poi in `EBP - 0x8` l'indirizzo della `vftable` e si invoca `__CxxThrowException@8` con i seguenti parametri:

Parametro	Valore
Parametro 1	19fc00
Parametro 2	47ee48

La funzione riprende poi dal ramo che avrebbe seguito se l'eccezione non fosse stata sollevata. Si procede inizializzando la struttura ottenuta come primo parametro, impostando il primo byte del buffer a 0, il campo `next_byte` a 0, e il campo `max_size` a 0xf. Questa struttura viene messa in `ECX` prima di invocare `FUN_0041fd90` con parametri:

Parametro	Valore
ECX	19fc30
Parametro 1	19fafc

## 9.26 FUN\_0041fd90

La funzione riceve due parametri:

Parametro	Valore
ECX	Indirizzo di una <code>sysroot_struct</code> appena allocata.
Parametro 1	??

La funzione prende il primo parametro e conta il numero di bytes successivi diversi da 0, che corrisponde quindi all'offset da aggiungere all'indirizzo base affinché punti al primo byte nullo. Questo valore viene messo sullo stack, seguito dall'indirizzo base (parametro 1). Viene poi invocata la funzione `FUN_0041f9d0` con 4 parametri:

Parametro	Valore
ECX	19fc38
EDX	19fafd
Parametro 3	19fafc
Parametro 4	3

## 9.27 FUN\_0041f9d0

La funzione riceve quattro parametri:

Parametro	Valore
Parametro 1: ECX	Indirizzo di una <code>sysroot_struct</code> appena allocata.
Parametro 2: EDX	Puntatore al byte successivo di parametro 3
Parametro 3	base address
Parametro 4	numero di bytes diversi da 0 a partire da base address

La funzione invoca subito `FUN_00419200` con parametri:

Parametro	Valore
ECX	10fc38
EDX	19fafd
Parametro 3	19fafc

Si controlla il byte meno significativo ritornato e, se è 0, procede invocando `FUN_0041bee0_struct_factory`. Se questa invocazione non ritorna 0, si procede confrontando il campo `max_len` con il valore 0x10, e EAX viene impostato conseguentemente:

$$EAX = \begin{cases} \text{dword ptr [ECX]} & \text{se max\_len} \geq 0x10 \\ ECX & \text{altrimenti} \end{cases}$$

Viene poi invocata la funzione `memcpy` per copiare 41fa51

## 9.28 FUN\_00419200

La funzione riceve tre parametri:

Parametro	Valore
Parametro 1: ECX	Indirizzo di una <code>sysroot_struct</code> appena allocata.
Parametro 2: EDX	Puntatore al byte successivo di parametro 3
Parametro 3	base address

Confronta il parametro 3 con il valore 0 e se sono diversi, mette in **EDX** il valore del campo **max\_len** della struttura in **ECX**. Questo valore viene confrontato con **0x10**; **EAX** viene impostato a:

$$\text{EAX} = \begin{cases} \text{dword ptr } [\text{ECX}] & \text{se max\_len} \geq 0x10 \\ \text{ECX} & \text{altrimenti} \end{cases}$$

Si confronta poi parametro 3 con **EAX**. Se il parametro 3 è minore, allora viene saltata gran parte della funzione. Il codice continua in questo caso azzerando il byte **AL** prima di ritornare.

## 9.29 `__CxxThrowException@8`

La funzione riceve due parametri:

Parametro	Valore
Parametro 1	???
Parametro 2	Indirizzo della <b>vftable</b>

Si salva in **EAX** il valore della **vftable** e in **ESI** l'indirizzo **0x47a3ac**. Si carica in **EDI** il valore di **[EBP - 0x20] = [19facc] = 0x208**. Si esegue poi `REP MOVSD word ptr [EDI], word ptr [ESI]` per **ECX = 8** volte: si copiano le 8 **dword** che partono da **ESI = 47a3ac** in **EDI = 19facc**.

Si memorizza il valore di **EAX = 19fc00** in **[EBP-8]**. Si sovrascrive **EAX** con l'indirizzo della **vftable** e lo si salva in **[EBP-4]**. Si effettuano poi i seguenti controlli per capire se mettere in **[EBP - 0xc]** il valore **1994000**; ciò accade se l'indirizzo della **vftable** non è 0 e l'AND logico tra il primo byte della **vftable** e 8 è diverso da 0.

In ogni caso, poi si continua caricando in **EAX** l'indirizzo di **[EBP - 0xc]**, cioè **19fae0**. Questo valore viene poi messo sullo stack, seguito dai valori in **[EBP - 0x10] = 3**, **[EBP - 0x1c] = 1** e **[EBP - 0x20] = e06d7363**. Si invoca poi `RaiseException` con questi parametri:

1. **ExceptionCode**: **e06d7363**
2. **ExceptionFlags**: **1**  $\equiv$  **EXCEPTION\_NONCONTINUABLE**
3. **nArgument**: 3. Numero di argomenti nel vettore di parametri successivo.
4. **pArgument**: **0019fae0**. Vettore di argomenti che vengono passati al gestore.

### 9.29.1 alcune librerie caricate

MulDiv, GetUserDefaultUILanguage, GetVersionExA, ExitProcess, GetModuleFileNameW, GetTmpPathW, LocalFree, MultiByteToWideChar, WideCharToMultiByte, GetLocaleInfoA, GetTempFileNameW, GetVolumeNameForVolumeMountPointA, GetWindowsDirectoryA, CreateProcessW, FindFirstFileW, FindClose, DeviceIoControl, LeaveCriticalSection, EnterCriticalSection, DeleteCriticalSection, InitializeCriticalSection, GetLogicalDrivers, GetDriveTypeW, GetVolumeInformationW, FindAtomA, FindNextFileW, GlobalFindAtomW, GlobalAddAtomA, AddAtomA, SetThreadPriority, GetCurrentThread, CopyFileW, GetUserDefaultLangID, GetSystemDefaultLangID, SetUnhandledExceptionFilter, SetErrorMode, ReadFile, WriteFile, FlushFileBuffers, GetFileSizeEx, SetFilePointer, SetFileTime, CreateFileW, DeleteFileW, MoveFileExW, GetSystemTimeAsFileTime...

## 10 Punti oscuri

- Che succede all'inizio quando sembra in attesa?
- Viene creato un nuovo thread?
- all'indirizzo 0x46dba9, dopo i vari controlli, c'è un test `this, this` su ghidra; che su *OllyDB* è `test ecx ecx`
- che succede ogni tanto? *OllyDB* non può continuare perché la memoria a un certo indirizzo è corrotta. Succede tipo quando spengo windows
- all'indirizzo 40c1fd c'è `jmp 40bf70` ma non vengono mostrati i blocchi successivi

## 11 Note

- `read_following_bit`: prende un solo parametro *non dallo stack*. È il registro ECX che contiene l'indirizzo 19f9b0, puntatore alla struttura dati
- `decode_bytes`: prende 3 parametri:
  1. il valore in ECX: rappresenta il numero di bit da leggere
  2. l'indirizzo della struttura dati: 19f9b0
  3. offset da sommare alla fine
- `ErrorMode = SEM_FAILCRITICALERRORS | SEM_NOGPFAULTERRORBOX | SEM_NOOPENFILEERRORBOX` all'indirizzo 42c17b
- `cmp a, b` fa sottrazione  $a - b$  e scarta il risultato, `test a, b` fa `AND` logico tra a e b e scarta il risultato
- RIPRENDI DA: *Ghidra*: 42b85c ; documento: 967
- Dare un nome alla funzione 4203a0, che invoca la funzione per copiare il buffer, copia gli altri 2 campi e modifica l'originale
- [il puntatore della classe c++ this è passato in ecx](#)
- 19fbe4 struttura dati di tipo `sysroot_struct`.
- [atom table](#)

## 12 to-do list

- vedere bene cosa fa fun 46d850, anche se scrive bytes potrebbe anche bastare come risposta.

## 13 Verifica