

# PROGETTO BUILDWEEK S12/L5

**CS0224**

---

Christopher Caruso

Maurizio Pietrangeli

Dario Calderone

Michele Pepe

Giulio Sorgente

Andrea Molla

17/05/2024

# SOMMARIO

TRACCIA	3
1. GIORNO 1	4
2. GIORNO 2	8
3. GIORNO 3	13
4. GIORNO 4	19
5. GIORNO 5	22

# TRACCIA

Il Malware da analizzare è nella cartella Build Week Unit 3 presente sul desktop della macchina virtuale dedicata.

Il testo della traccia di ogni giorno è riportato nell'intestazione del relativo capitolo.

## 1. GIORNO 1

Con riferimento al file eseguibile Malware\_Build\_Week\_U3, rispondere ai seguenti quesiti utilizzando i tool e le tecniche apprese nelle lezioni teoriche:

- Quanti **parametri** sono passati alla funzione Main()?
- Quante **variabili** sono dichiarate all'interno della funzione Main()?

I **parametri** passati alla funzione Main() sono 3: argc, argv, envp.

Questo si può dedurre dall'offset positivo della dichiarazione riportata nell'immagine sottostante (parametri evidenziati in giallo).

Poiché, invece, le dichiarazioni con offset negativo sono 5 (variabili locali evidenziate in arancione), le **variabili** rilevate sono: hModule, Data, var\_117, var\_8, var\_4.

Per comprendere il tipo di offset, è sufficiente verificare se il valore nella dichiarazione contenuta all'interno del codice è positivo o negativo.

Il tipo di offset, relativo alla porzione di memoria nello stack, indica in che direzione si effettua lo spostamento del puntatore per l'accesso al dato rispetto alla base dello stack.

```
.text:004011D0 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:004011D0 _main          proc near          ; CODE XREF: start+AF↓p
.text:004011D0
.text:004011D0 hModule      = dword ptr -11Ch
.text:004011D0 Data        = byte ptr -118h
.text:004011D0 var_117     = byte ptr -117h
.text:004011D0 var_8       = dword ptr -8
.text:004011D0 var_4       = dword ptr -4
.text:004011D0 argc        = dword ptr 8
.text:004011D0 argv        = dword ptr 8Ch
.text:004011D0 envp        = dword ptr 10h
```

- Quali **sezioni** sono presenti all'interno del file eseguibile?  
Descrivete brevemente almeno 2 di quelle identificate.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00005646	00001000	00006000	00001000	00000000	00000000	0000	0000	60000020
.rdata	000009AE	00007000	00001000	00007000	00000000	00000000	0000	0000	40000040
.data	00003EA8	00008000	00003000	00008000	00000000	00000000	0000	0000	C0000040
.rsrc	00001A70	0000C000	00002000	0000B000	00000000	00000000	0000	0000	40000040

Le **sezioni** presenti all'interno del malware sono 4:

- **.text** → contiene le istruzioni che la CPU esegue quando viene avviato il software. È l'unica sezione di un file eseguibile che viene eseguita dalla CPU, le altre contengono solo dati o informazioni a supporto;
- **.rdata** → include informazioni su librerie e funzioni importate ed esportate dall'eseguibile;
- **.data** → contiene dati e variabili globali del programma eseguibile;
- **.rsrc** → include risorse utilizzate dall'eseguibile come ad esempio icone, immagini, menu e stringhe che non sono parte dell'eseguibile stesso.

- Quali **librerie** importa il Malware? Per ognuna delle librerie importate, fate delle **ipotesi** sulla base della sola analisi statica delle funzionalità che il Malware potrebbe implementare. Utilizzate le funzioni che sono richiamate all'interno delle librerie per supportare le vostre ipotesi.

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000769E	N/A	000074EC	000074F0	000074F4	000074F8	000074FC
szAnsi	(nFunctions)	Dword		Dword	Dword	Dword
KERNEL32.dll	51	00007534	00000000	00000000	0000769E	0000700C
ADVAPI32.dll	2	00007528	00000000	00000000	000076D0	00007000

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
00007632	00007632	0295	SizeofResource
00007644	00007644	01D5	LockResource
00007654	00007654	01C7	LoadResource
00007622	00007622	02BB	VirtualAlloc
00007674	00007674	0124	GetModuleFileNameA
0000768A	0000768A	0126	GetModuleHandleA
00007612	00007612	00B6	FreeResource
00007664	00007664	00A3	FindResourceA
00007604	00007604	001B	CloseHandle
000076DE	000076DE	00CA	GetCommandLineA
000076F0	000076F0	0174	GetVersion

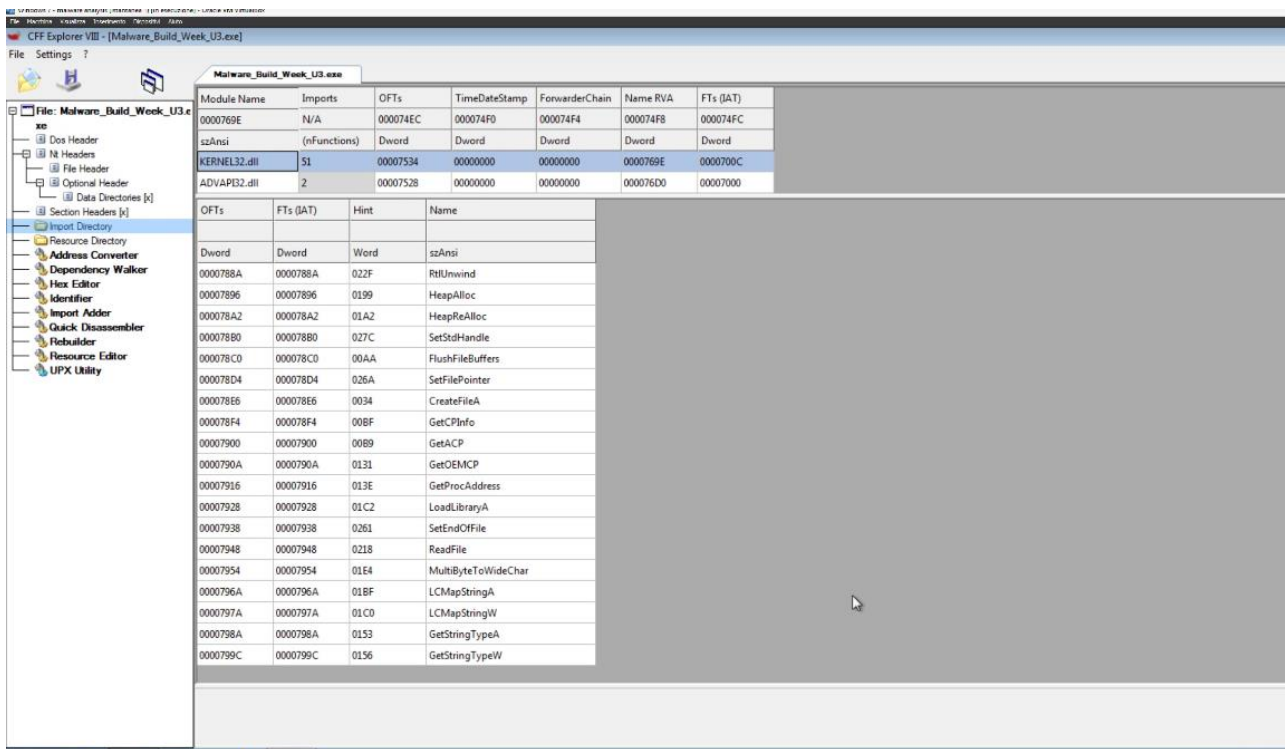
Le **librerie** importate dal malware sono 2:

- KERNEL32.dll → contiene le funzioni principali per interagire con il sistema operativo, ad esempio manipolazione di file e gestione della memoria;
- ADVAPI32.dll → contiene le funzioni per interagire con i servizi ed i registri del sistema operativo (probabilmente usate per ottenere persistenza).

Tra le funzionalità comuni dei malware analizzate a lezione sono state individuate le seguenti funzioni: FindResource(), LoadResource(), LockResource() e SizeOfResource().

Queste sono funzioni tipicamente adottate dai **dropper** che usano tali funzioni per localizzare all'interno della sezione "risorse" il malware da estrarre, e

successivamente da caricare in memoria per l'esecuzione immediata o da salvare sul disco per l'esecuzione futura.



Vengono inoltre utilizzate altre funzionalità come:

- CreateFile(), WriteFile() → tipicamente usate da un dropper per il salvataggio su disco;
- LoadLibrary() → tipicamente usata per caricare librerie a runtime, dunque non è possibile analizzare il completo funzionamento del malware con la sola analisi statica.

**Ipotesi:** dalle considerazioni effettuate in precedenza sembra dunque trattarsi di un **dropper**.

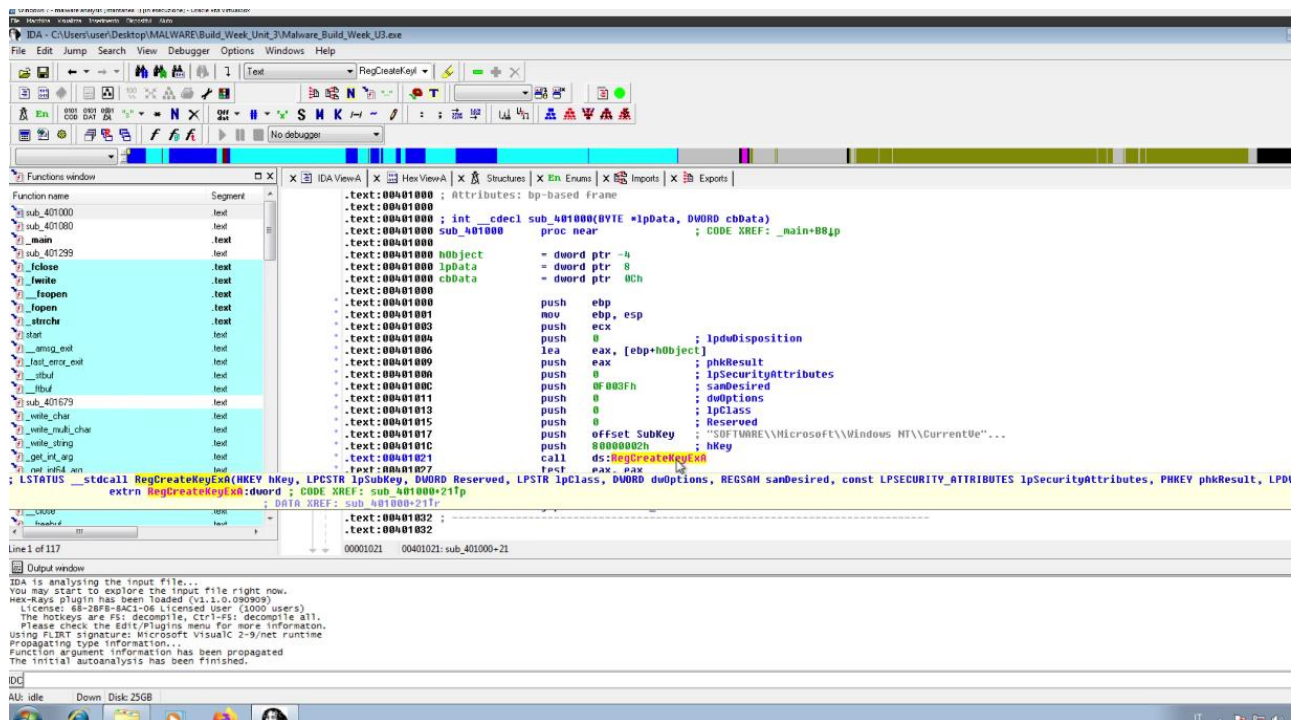


## 2. GIORNO 2

Con riferimento al Malware in analisi, spiegare:

- Lo scopo della **funzione** chiamata alla locazione di memoria 00401021;

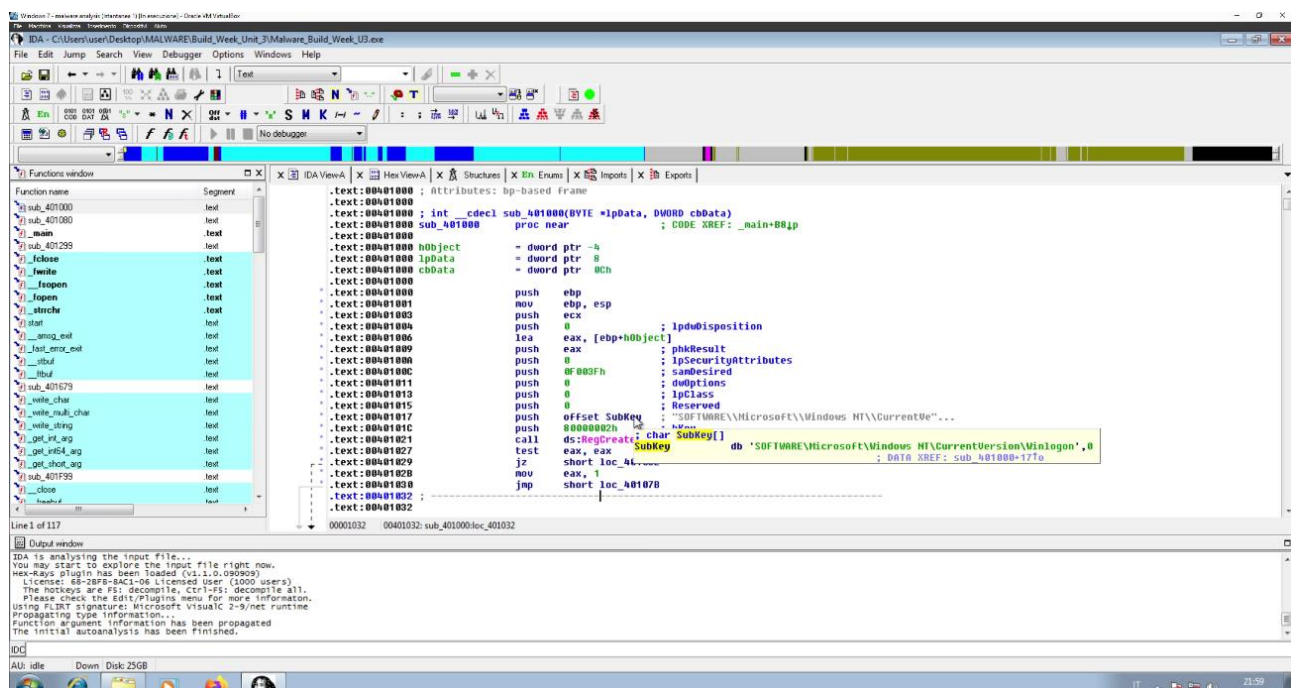
Nella locazione di memoria 00401021 viene chiamata la **funzione** RegCreateKeyExA. Lo scopo di questa funzione è quello di creare o editare una chiave di registro.





- Come vengono passati i **parametri** alla funzione alla locazione 00401021;

Dalla locazione di memoria 00401000 sono presenti le istruzioni per la creazione dello stack e a seguire sono presenti le istruzioni *push* per il caricamento sullo stack dei parametri della funzione. Dunque, il passaggio dei **parametri** avviene tramite stack.



- Che **oggetto** rappresenta il parametro alla locazione 00401017;

Nella locazione di memoria 00401017 è presente la SubKey contenente il path "SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon". L'**oggetto** in esame è dunque una chiave di registro che è poi passata tramite stack alla funzione *RegCreateKeyExA*.

- Il **significato** delle istruzioni comprese tra gli indirizzi 00401027 e 00401029 (se serve, valutate un'altra o altre due righe assembly);

```

.text:00401027      test     eax, eax
.text:00401029      jz       short loc_401032
.text:0040102B      mov     eax, 1
.text:00401030      jmp     short loc_40107B
.text:00401032      ; -----
.text:00401032      loc_401032:      mov     ecx, [ebp+cbData] ; CODE XREF: sub_401000+291j
.text:00401032      push    ecx        ; cbData
.text:00401035      mov     edx, [ebp+lpData]
.text:00401036      push    edx        ; lpData
.text:00401039      push    1          ; dwType
.text:0040103A      push    0          ; Reserved
.text:0040103C      push    offset ValueName ; "GinaDLL"
.text:0040103E      mov     eax, [ebp+hObject]
.text:00401043      push    eax        ; hKey
.text:00401046      call    ds:RegSetValueExA
.text:00401047      test    eax, eax
.text:0040104D      jz       short loc_401062
.text:00401051      mov     ecx, [ebp+hObject]
.text:00401054      push    ecx        ; hObject
.text:00401055      call    ds:CloseHandle
.text:0040105B      mov     eax, 1
.text:00401060      jmp     short loc_40107B
.text:0040107B      loc_40107B:      ; CODE XREF: sub_401000+307j
.text:0040107B      ; sub_401000+607j
.text:0040107B      mov     esp, ebp
.text:0040107D      pop     ebp
.text:0040107E      retn
.text:0040107E      sub_401000      endp

```

L'istruzione condizionale «test» (presente nella locazione 00401027) è simile all'istruzione AND, ma a differenza di essa non modifica il contenuto degli operandi.

Tuttavia, modifica il flag ZF (zero flag) del registro EFLAGS, che viene settato ad 1 se e solo se il risultato dell'AND è 0.

Viene, quindi, usata per controllare se un valore è uguale a zero o meno.

In questo caso, quindi, verifica se eax è uguale a zero.

L'istruzione jz serve per effettuare un salto alla locazione di memoria indicata, nel caso in cui il flag ZF sia uguale a 1.

**Queste due istruzioni corrispondono ad un costrutto IF**, nel quale viene verificato se il valore del registro eax sia uguale a 0, in caso affermativo il risultato della AND è 0, il flag ZF viene settato a 1 e viene effettuato il salto alla locazione 401032.

Le istruzioni che seguono il jump condizionale jz corrispondono ad un **else**, poiché vengono eseguite se la condizione precedente non si verifica.

Se, quindi, il jump condizionale jz non viene effettuato, entra in esecuzione l'istruzione mov che inserisce nel registro eax il valore 1; seguita da un jump non

condizionale *jmp* che salta in qualsiasi caso alla locazione 40107B (pulizia stack e restituzione controllo alla funzione chiamante).

- Con riferimento all'ultimo quesito, tradurre il codice Assembly nel corrispondente costrutto C;

## Syntax

```
C++
LSTATUS RegSetValueExA(
    [in]      HKEY      hKey,
    [in, optional] LPCSTR lpValueName,
    [in]      DWORD     Reserved,
    [in]      DWORD     dwType,
    [in]      const BYTE *lpData,
    [in]      DWORD     cbData
);
```

```
if( hObject == NULL ){ //hObject è la key che viene caricata in eax

    RegSetValueExA(hObject, ValueName ,Reserved, dwType, val_lpData, val_cbData);
    //al posto dei nomi dei parametri vengono passati i valori corrispondenti (val_)
    //ad esempio dwType = 1 e Reserved = 0, ValueName = GinaDLL
    ...

    CloseHandle(hObject);

    ...
}
else{
    ...
}
```

In sintesi, il costrutto *if* che corrisponde al codice assembly verifica che la chiave *hKey* non sia settata e nel caso esegue la chiamata a funzione *RegSetValueExA* passando i parametri sullo stack (viene eseguito tutto il codice presente nella locazione di memoria del jump condizionale *jz*).



### 3. GIORNO 3

Riprendete l'analisi del codice, analizzando le routine tra le locazioni di memoria 00401080 e 00401128:

```
.text:00401080 ; ----- SUBROUTINE -----
.text:00401080 ; Attributes: bp-based frame
.text:00401080 ; int __cdecl sub_401080(HMODULE hModule)
.text:00401080 sub_401080      proc near          ; CODE XREF: _main+3F↓p
.text:00401080 hResData      = dword ptr -18h
.text:00401080 hResInfo     = dword ptr -14h
.text:00401080 Count       = dword ptr -10h
.text:00401080 var_C        = dword ptr -0Ch
.text:00401080 Str         = dword ptr -8
.text:00401080 File        = dword ptr -4
.text:00401080 hModule      = dword ptr 8
.text:00401080
.text:00401080      push    ebp
.text:00401081      mov     ebp, esp
.text:00401083      sub     esp, 18h
.text:00401086      push    esi
.text:00401087      push    edi
.text:00401088      mov     [ebp+hResInfo], 0
.text:0040108F      mov     [ebp+hResData], 0
.text:00401096      mov     [ebp+Str], 0
.text:0040109D      mov     [ebp+Count], 0
.text:004010A4      mov     [ebp+var_C], 0
.text:004010AB      cmp     [ebp+hModule], 0
```

```
.text:004010AB      cmp     [ebp+hModule], 0
.text:004010AF      jnz     short loc_4010B8
.text:004010B1      xor     eax, eax
.text:004010B3      jmp     loc_4011BF
.text:004010B8 ; -----
.text:004010B8 loc_4010B8:          ; CODE XREF: sub_401080+2F↑j
.text:004010B8      mov     eax, lpType
.text:004010BD      push    eax          ; lpType
.text:004010BE      mov     ecx, lpName
.text:004010C4      push    ecx          ; lpName
.text:004010C5      mov     edx, [ebp+hModule]
.text:004010C8      push    edx          ; hModule
.text:004010C9      call    ds:FindResourceA
.text:004010CF      mov     [ebp+hResInfo], eax
.text:004010D2      cmp     [ebp+hResInfo], 0
.text:004010D6      jnz     short loc_4010DF
.text:004010D8      xor     eax, eax
.text:004010DA      jmp     loc_4011BF
.text:004010DF ; -----
.text:004010DF loc_4010DF:          ; CODE XREF: sub_401080+56↑j
.text:004010DF      mov     eax, [ebp+hResInfo]
.text:004010E2      push    eax          ; hResInfo
.text:004010E3      mov     ecx, [ebp+hModule]
.text:004010E6      push    ecx          ; hModule
.text:004010E7      call    ds:LoadResource
.text:004010ED      mov     [ebp+hResData], eax
```



```

.text:004010F0      cmp     [ebp+hResData], 0
.text:004010F4      jnz     short loc_4010FB
.text:004010F6      jmp     loc_4011A5
;-----
.text:004010FB      loc_4010FB:
.text:004010FB      mov     edx, [ebp+hResData] ; CODE XREF: sub_401080+74tj
.text:004010FE      push    edx ; hResData
.text:004010FF      call    ds:LockResource
.text:00401105      mov     [ebp+Str], eax
.text:00401108      cmp     [ebp+Str], 0
.text:0040110C      jnz     short loc_401113
.text:0040110E      jmp     loc_4011A5
;-----
.text:00401113      loc_401113:
.text:00401113      mov     eax, [ebp+hResInfo] ; CODE XREF: sub_401080+8Ctj
.text:00401116      push    eax ; hResInfo
.text:00401117      mov     ecx, [ebp+hModule]
.text:0040111A      push    ecx ; hModule
.text:0040111B      call    ds:SizeofResource
.text:00401121      mov     [ebp+Count], eax
.text:00401124      cmp     [ebp+Count], 0
.text:00401128      ja      short loc_40112C

```

- Qual è il **valore del parametro** «ResourceName» passato alla funzione FindResourceA();

La funzione FindResourceA() riceve come parametri *hModule*, *lpName*, *lpType*.

```

;-----
.text:00401088      loc_401088:
.text:00401088      mov     eax, lpType ; CODE XREF: sub_401080+2Ftj
.text:0040108B      push    eax ; lpType
.text:0040108D      mov     ecx, lpName ; lpName
.text:0040108E      push    ecx ; lpName
.text:004010C4      mov     edx, [ebp+hModule]
.text:004010C8      push    edx ; hModule
.text:004010C9      call    ds:FindResourceA
.text:004010CF      mov     [ebp+hResInfo], eax
;-----
; HRSRC __stdcall FindResourceA(HMODULE hModule, LPCSTR lpName, LPCSTR lpType)
; extrn FindResourceA:duword ; CODE XREF: sub_401080+49tj
; DATA XREF: sub_401080+49tj
;-----
.text:004010DA      jmp     loc_4011BF

```

Cercando nella sezione .data il valore di *lpName* è *TGAD*.

```

Name: lpName
Options: Windows Help
No debugger

IDA View: X IDA View A X Hex View A X Structures X En Enms X Imports X Exports
segment
* .data:0040802E db 0
* .data:0040802F db 0
* .data:00408030 ; LPCSTR lpType
* .data:00408030 dd offset aBinary ; DATA XREF: sub_401080:loc_401088tj
* .data:00408030 ; "BINARY"
* .data:00408034 ; LPCSTR lpName
* .data:00408034 dd offset aTgad ; DATA XREF: sub_401080+3Etj
* .data:00408034 ; "TGAD"
* .data:00408038 aTgad db 'TGAD',0 ; DATA XREF: .data:lpNamefo
* .data:0040803D align 10h
* .data:00408040 aBinary db 'BINARY',0 ; DATA XREF: .data:lpTypefo
* .data:00408047 align 4
* .data:00408048 aRi db 'RI',00h,0 ; DATA XREF: sub_401080:loc_401062to

```

- Il susseguirsi delle chiamate di funzione che effettua il Malware in questa sezione di codice l'abbiamo visto durante le lezioni teoriche. **Che funzionalità sta implementando il Malware?**

Tra le funzionalità comuni dei malware analizzate a lezione sono state individuate le seguenti funzioni: `FindResource()`, `LoadResource()`, `LockResource()` e `SizeOfResource()`.

Queste sono funzioni tipicamente adottate dai **dropper** che usano tali funzioni per localizzare all'interno della sezione "risorse" il file da estrarre, e successivamente da caricare in memoria per l'uso immediato o da salvare sul disco per l'uso futuro.

- È possibile identificare questa funzionalità utilizzando l'analisi statica basica? (dal giorno 1 in pratica);

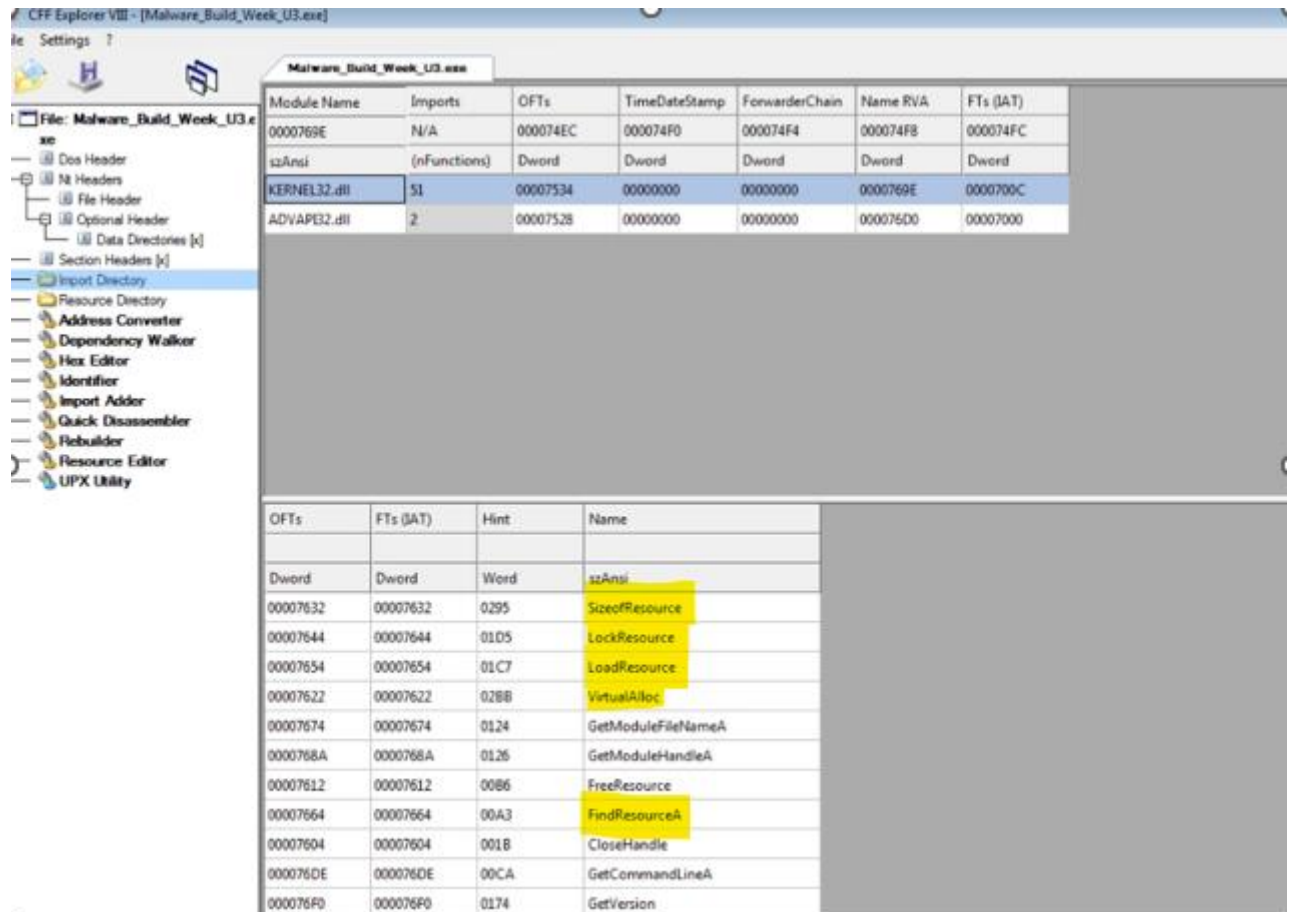
Sì, è possibile perché lo abbiamo individuato anche nelle funzioni caricate tramite CFF Explorer.



- In caso di risposta affermativa, elencare le evidenze a supporto.

Come si vede dalla seguente immagine le funzioni elencate sono già rilevabili con CFF Explorer.

Le considerazioni effettuate nel giorno 1 chiariscono già la possibile natura del malware e l'analisi con IDA è un'ulteriore validazione di tale ipotesi.



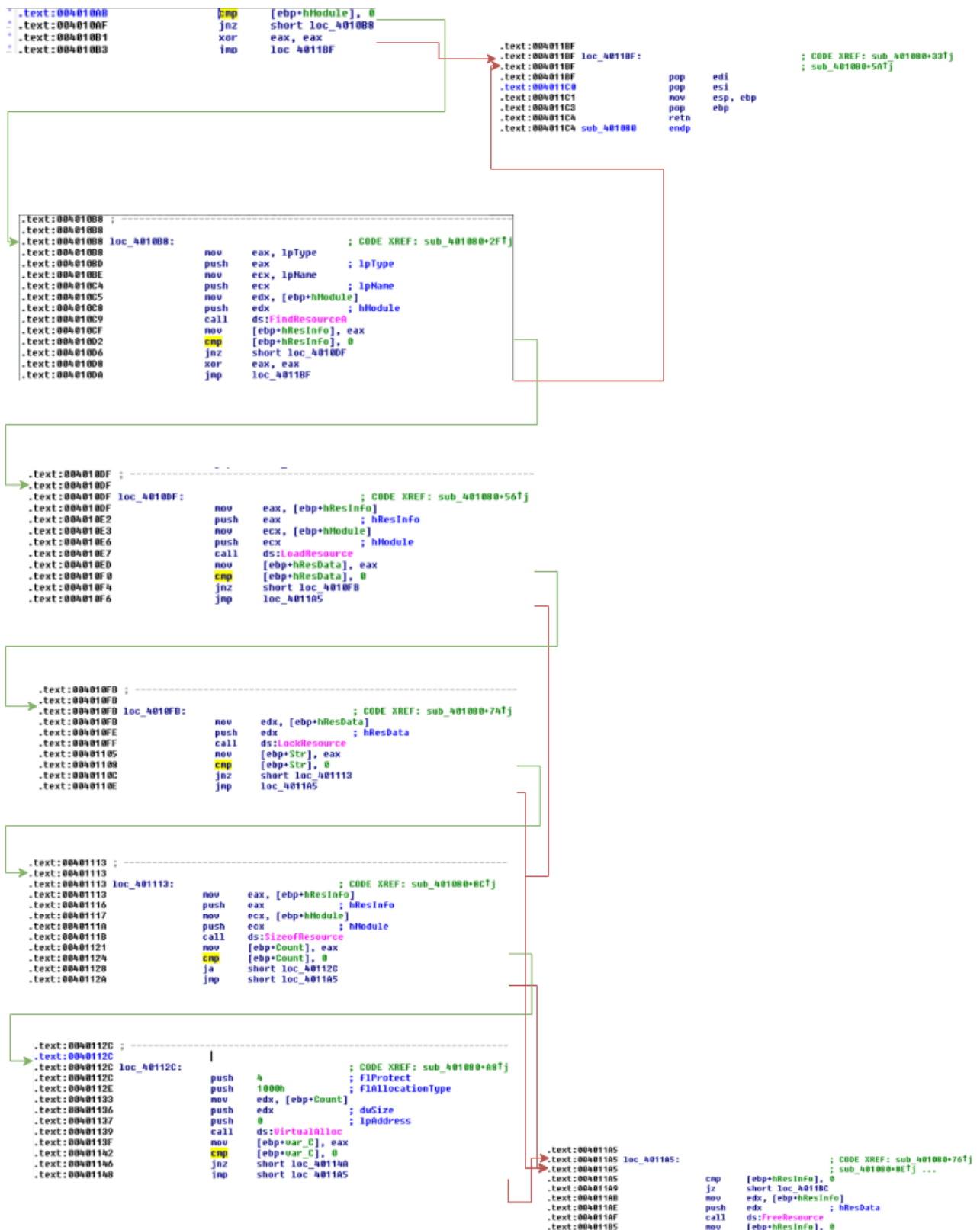
Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (JAT)
0000769E	N/A	000074EC	000074F0	000074F4	000074F8	000074FC
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	51	00007534	00000000	00000000	0000769E	0000700C
ADVAPI32.dll	2	00007528	00000000	00000000	000076D0	00007000

OFTs	FTs (JAT)	Hint	Name
Dword	Dword	Word	szAnsi
00007632	00007632	0295	SizeofResource
00007644	00007644	01D5	LockResource
00007654	00007654	01C7	LoadResource
00007622	00007622	028B	VirtualAlloc
00007674	00007674	0124	GetModuleFileNameA
0000768A	0000768A	0126	GetModuleHandleA
00007612	00007612	0086	FreeResource
00007664	00007664	00A3	FindResourceA
00007604	00007604	001B	CloseHandle
000076DE	000076DE	00CA	GetCommandLineA
000076F0	000076F0	0174	GetVersion

- Entrambe le funzionalità principali del Malware viste finora sono richiamate all'interno della funzione Main().

**Disegnare un diagramma di flusso** (inserite all'interno dei box solo le informazioni circa le funzionalità principali) che comprenda le 3 funzioni.



Nella precedente immagine è possibile vedere un diagramma di flusso con approssimazione alle sole parti interessate dalle funzionalità del dropper (in questo caso le frecce verdi descrivono i salti condizionali e quelle rosse i salti non condizionali, non è stata quindi usata la linea adottata da IDA per non rendere il diagramma troppo elaborato).

Dopo il primo costrutto condizionale (*cmp + jnz*) alla locazione di memoria 004010AB, al verificarsi della condizione viene effettuato un salto alla locazione 004010B8 contenente la funzione *FindResource()*, altrimenti viene effettuato un salto non condizionale (*jmp*) alla locazione 004011BF per la pulizia dello stack e la restituzione del controllo alla funzione chiamante.

Nel caso in cui si entri nella locazione di memoria della funzione *FindResource()*, viene effettuato un salto condizionale (*cmp + jnz*) che porta alla locazione di memoria della funzione *LoadResource()*. Se le condizioni del controllo non vengono verificate, viene effettuato un salto non condizionale (*jmp*) alla locazione 004011BF per la pulizia dello stack e la restituzione del controllo alla funzione chiamante.

Dalla locazione di memoria della funzione *LoadResource()* seguono una serie di costrutti if (*cmp + jnz* o *ja*) che servono per le call function a catena relative alle funzioni *LockResource()*, *SizeOfResource()*, *VirtualAlloc()*, che prevedono a loro volta un salto non condizionale (*jmp*) per entrare nella locazione di memoria della funzione *FreeResource()*.

Questo flusso descrive perfettamente il comportamento tipico di un **dropper**.

## 4. GIORNO 4

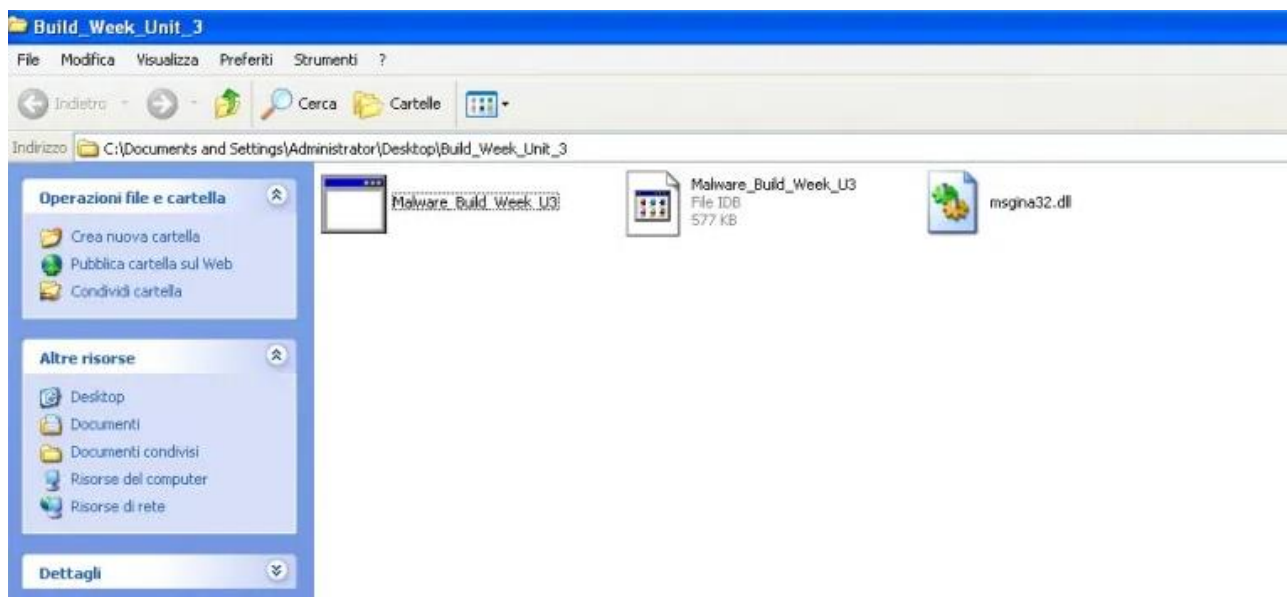
Preparate l'ambiente ed i tool per l'esecuzione del Malware (suggerimento: avviate principalmente Process Monitor ed assicurate di eliminare ogni filtro cliccando sul tasto «reset» quando richiesto in fase di avvio).

Eseguite il Malware, facendo doppio click sull'icona dell'eseguibile.

Da una ricerca online\* per capire la natura della dll Gina, è emerso che la manipolazione è permessa solo da sistemi operativi Windows antecedenti a Vista. Pertanto, l'esecuzione dei test è stata effettuata su Windows XP.

\*<https://stackoverflow.com/questions/2657491/what-is-a-gina-dll>

- Cosa notate all'interno della cartella dove è situato l'eseguibile del Malware? Spiegate cosa è avvenuto, unendo le evidenze che avete raccolto finora per rispondere alla domanda.



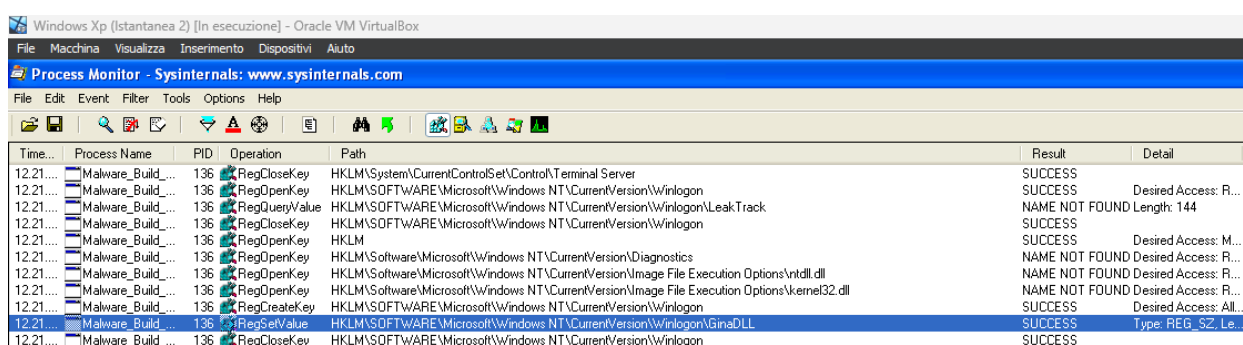
Come si vede dall'immagine sopra viene caricato un file *msgina32.dll*.

Rispetto a quanto emerso in precedenza sembra essere confermata l'ipotesi di un **dropper** poiché probabilmente questo file verrà caricato a sistema essendo una dll.

Analizzate ora i risultati di Process Monitor (consiglio: utilizzate il filtro come in figura sotto per estrarre solo le modifiche apportate al sistema da parte del Malware). Fate click su «ADD» poi su «Apply» come abbiamo visto nella lezione teorica.

Filtrate includendo solamente l'attività sul **registro windows**.

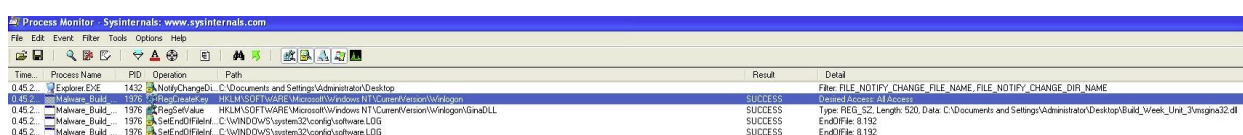
- Quale chiave di registro viene creata?
- Quale valore viene associato alla chiave di registro creata?



Windows Xp (Istantanea 2) [In esecuzione] - Oracle VM VirtualBox

Process Monitor - Sysinternals: www.sysinternals.com

Time...	Process Name	PID	Operation	Path	Result	Detail
12.21...	Malware_Build...	136	RegCloseKey	HKLM\System\CurrentControlSet\Control\Terminal Server	SUCCESS	
12.21...	Malware_Build...	136	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	Desired Access: R...
12.21...	Malware_Build...	136	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\LeakTrack	NAME NOT FOUND	Length: 144
12.21...	Malware_Build...	136	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	
12.21...	Malware_Build...	136	RegOpenKey	HKLM	SUCCESS	Desired Access: M...
12.21...	Malware_Build...	136	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Diagnostics	NAME NOT FOUND	Desired Access: R...
12.21...	Malware_Build...	136	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ntdll.dll	NAME NOT FOUND	Desired Access: R...
12.21...	Malware_Build...	136	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\kernel32.dll	NAME NOT FOUND	Desired Access: R...
12.21...	Malware_Build...	136	RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	Desired Access: All...
12.21...	Malware_Build...	136	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL	SUCCESS	Type: REG_SZ, Le...
12.21...	Malware_Build...	136	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	



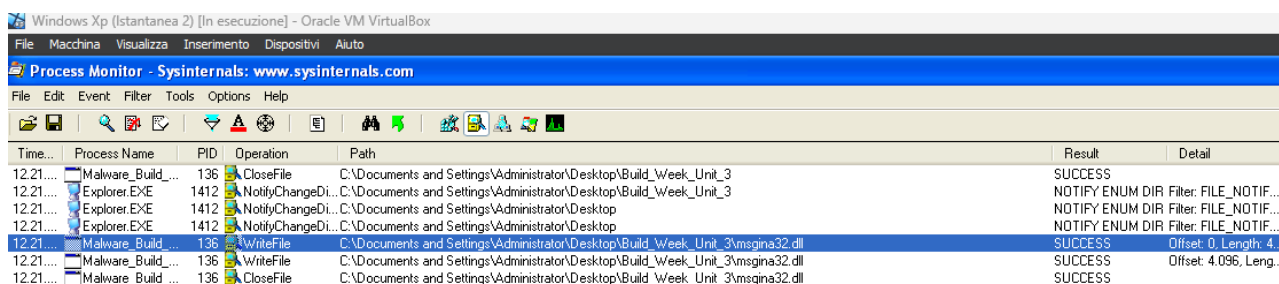
Process Monitor - Sysinternals: www.sysinternals.com

Time...	Process Name	PID	Operation	Path	Result	Detail
0.45.2...	Explorer.exe	1432	NotifyChangeDir	C:\Documents and Settings\Administrator\Desktop		Filter: FILE_NOTIFY_CHANGE_FILE_NAME, FILE_NOTIFY_CHANGE_DIR_NAME
0.45.2...	Malware_Build...	1376	RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	Desired Access: All Access
0.45.2...	Malware_Build...	1376	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL	SUCCESS	Type: REG_SZ, Length: 520, Data: C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll
0.45.2...	Malware_Build...	1376	SetEndOfFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 8192
0.45.2...	Malware_Build...	1376	SetEndOfFile	C:\WINDOWS\system32\config\software.LOG	SUCCESS	EndOfFile: 8192

Dalle precedenti immagini è evidente che il malware provi a **manipolare la chiave di registro** precedentemente individuata (HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\GinaDLL) provando a valorizzare il path con quello del file *msgina32.dll* (C:\Documents and Settings\Administrator\Desktop\Build\_Week\_Unit\_3\msgina32.dll).

Passate ora alla visualizzazione dell'attività sul **File System**.

- Quale chiamata di sistema ha modificato il contenuto della cartella dove è presente l'eseguibile del Malware?



Time...	Process Name	PID	Operation	Path	Result	Detail
12.21....	Malware_Build_...	136	CloseFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	SUCCESS	
12.21....	Explorer.EXE	1412	NotifyChangeDi...	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	NOTIFY ENUM DIR Filter: FILE_NOTIF...	
12.21....	Explorer.EXE	1412	NotifyChangeDi...	C:\Documents and Settings\Administrator\Desktop	NOTIFY ENUM DIR Filter: FILE_NOTIF...	
12.21....	Explorer.EXE	1412	NotifyChangeDi...	C:\Documents and Settings\Administrator\Desktop	NOTIFY ENUM DIR Filter: FILE_NOTIF...	
12.21....	Malware_Build_...	136	WriteFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS	Offset: 0, Length: 4...
12.21....	Malware_Build_...	136	WriteFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS	Offset: 4,096, Leng...
12.21....	Malware_Build_...	136	CloseFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS	

Dalla precedente Immagine è evidente che il malware tramite funzione *WriteFile* **carichi la dll nel path precedentemente indicato** dell'utente corrente (*Administrator*).

Unite tutte le informazioni raccolte fin qui sia dall'analisi statica che dall'analisi dinamica per delineare il funzionamento del Malware.

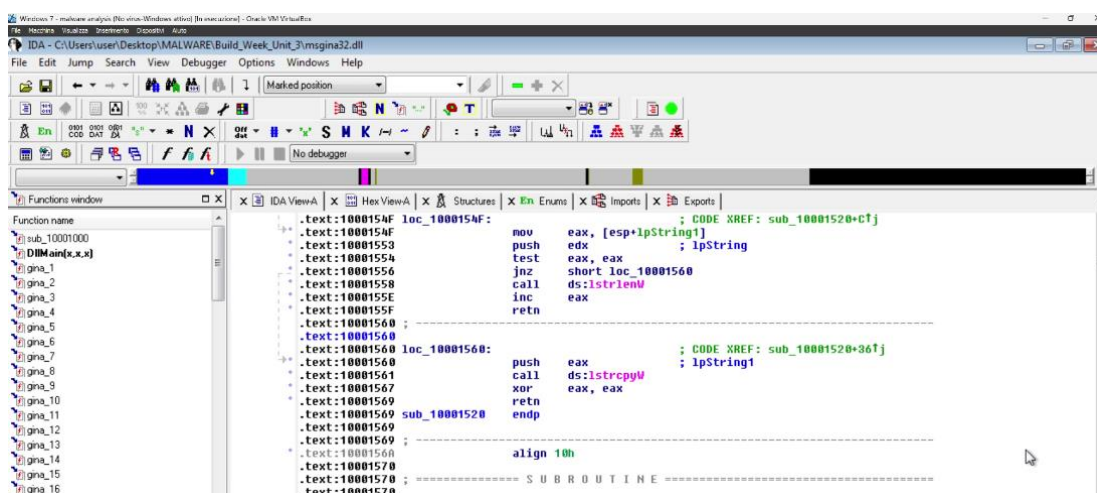
Dalle analisi svolte finora quello che si evince è che il malware cerca di modificare il meccanismo di autenticazione di Windows manipolando la DLL Gina. Questo eseguibile che potrebbe apparire innocuo nasconde (def. *trojan*), quindi, un comportamento malevolo che altera il meccanismo di autenticazione di Windows.

## 5. GIORNO 5

GINA (Graphical identification and authentication) è un componente lecito di Windows che permette l'autenticazione degli utenti tramite interfaccia grafica – ovvero permette agli utenti di inserire **username** e **password** nel classico riquadro Windows, come quello in figura a destra che usate anche voi per accedere alla macchina virtuale.

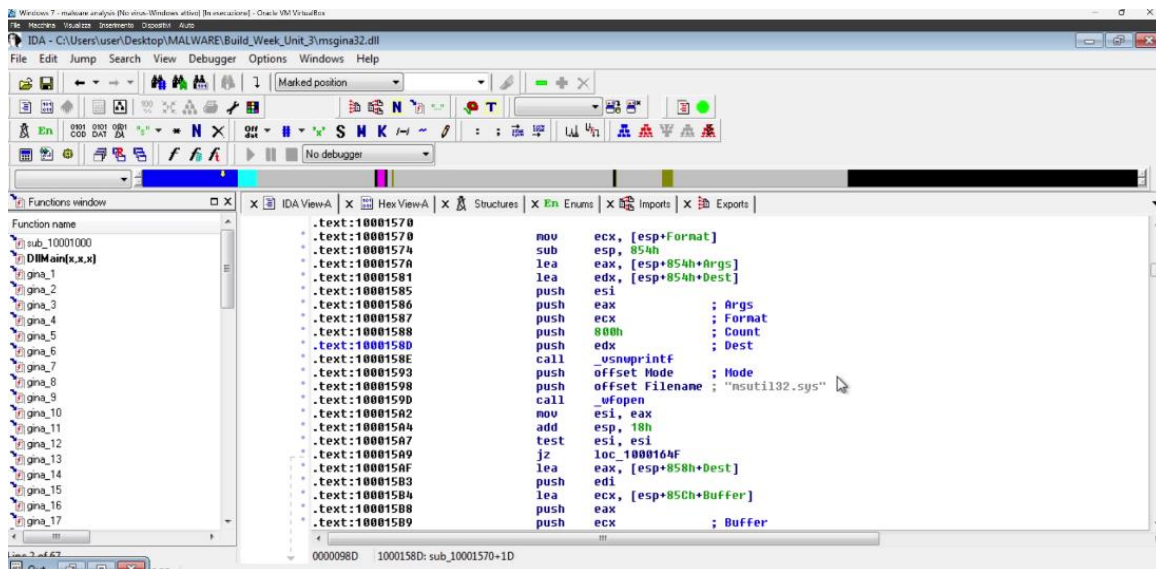


- Cosa può succedere se il file .dll lecito viene sostituito con un file .dll malevolo, che intercetta i dati inseriti?

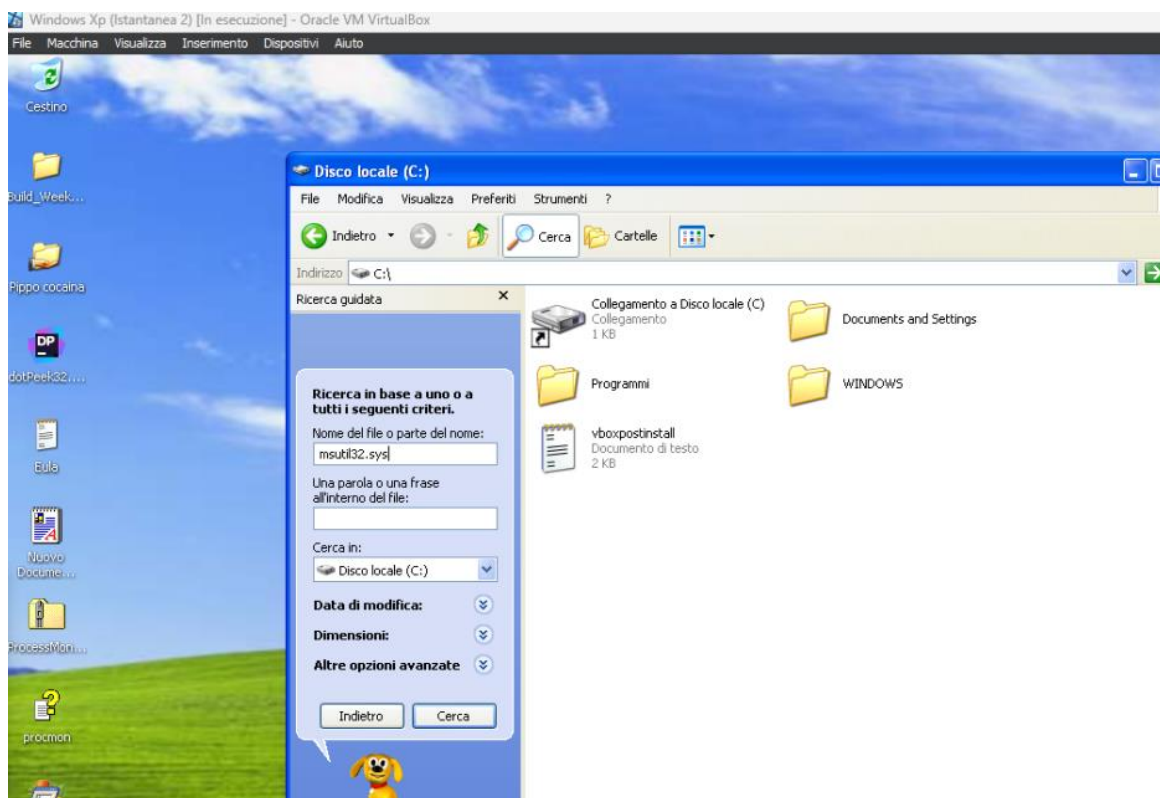


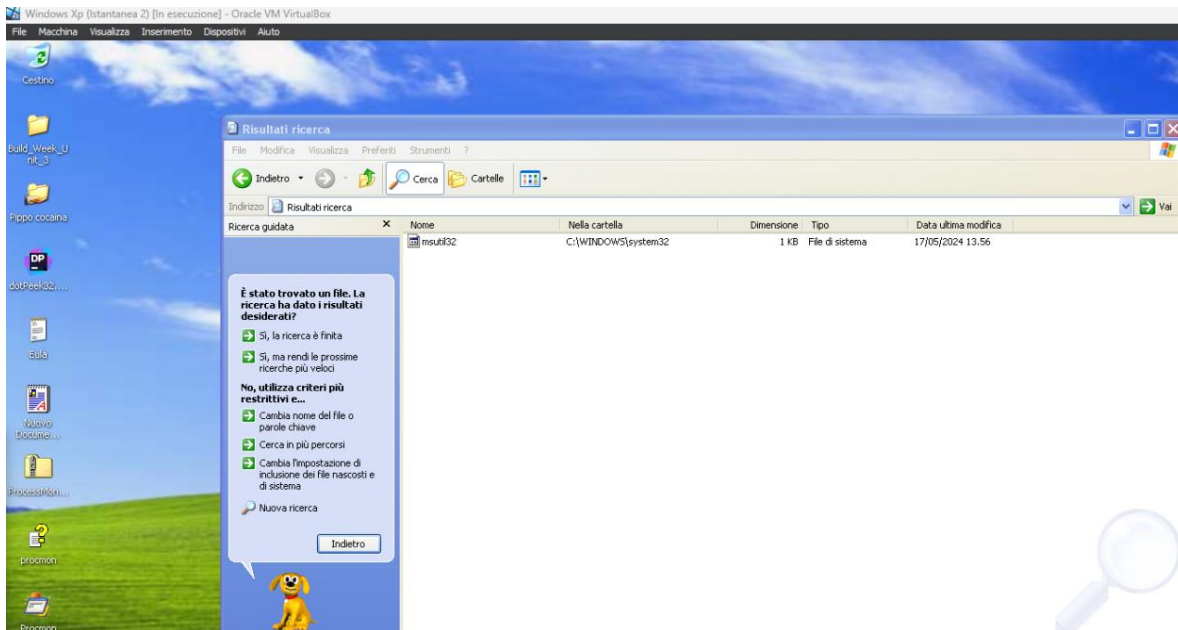


Analizzando il Main() della DLL *msgina32.dll* con IDA emerge l'utilizzo delle funzioni *lstrlenW()* e *lstrcpyW()* che servono ad operare su stringhe. La funzione *lstrcpyW()* serve ad aprire un buffer per copiare il contenuto della stringa puntata.

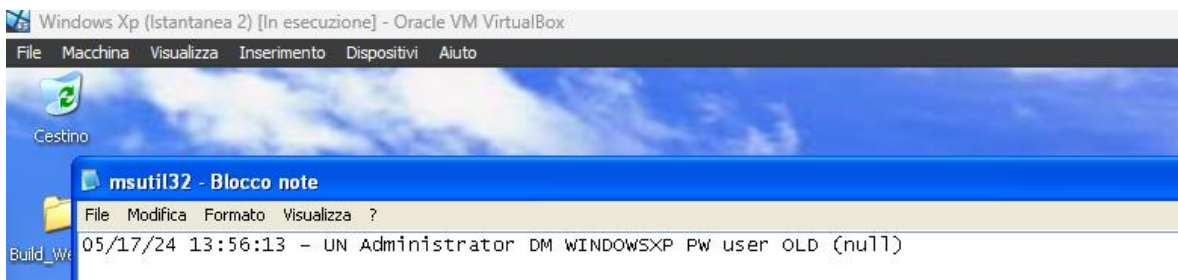


Effettuando una ricerca da OS col nome del file rilevato, è stato trovato un file contenente i tentativi di accesso/login al sistema.





Aperto il file, quindi, vengono visualizzati gli accessi come da immagine seguente.



Da questo file emerge che per l'utente Administrator la password non è impostata.

Sulla base della risposta sopra, delineate il profilo del Malware e delle sue funzionalità.

Il malware in esame è dunque un **trojan** che nasconde le funzionalità di un **dropper** che sostituisce il file *msgina* originale di Windows con il file *msgina32.dll*. La sostituzione di questo file ha lo scopo di alterare il meccanismo di login per intercettare le credenziali degli utenti sull' host nella fase di autenticazione tramite stream sul buffer, salvando il log nel file *msutil32.sys*.

Unite tutti i punti per creare un grafico che ne rappresenti lo scopo ad alto livello.

