

REPORT – BUILD WEEK 3

MALWARE ANALYSIS

TEAM 1

Floriana Feminò
Francesco Persichetti
Fabio Herrera
Lorenzo Scorbaioli
Emanuel Pollidoro
Andrea Cuore
Orlando Tangari

GIORNO 1

1. Quanti parametri sono passati alla funzione Main()? Quante variabili sono dichiarate all'interno della funzione Main()?
2. Quali sezioni sono presenti all'interno del file eseguibile? Descrivete brevemente almeno 2 di quelle identificate
3. 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.

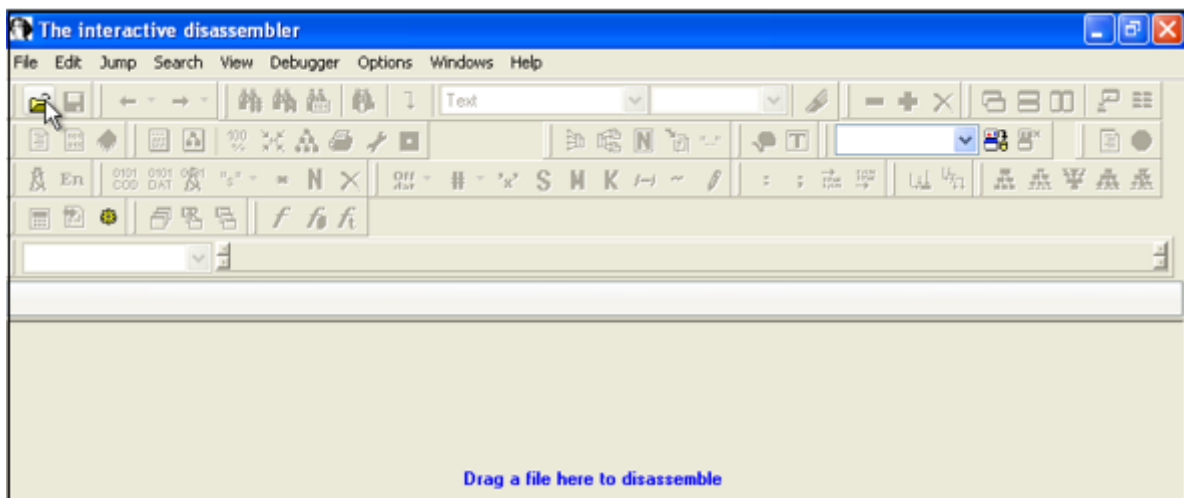
1. Parametri e variabili nella funzione Main()

Il Malware da analizzare è nella cartella Build_Week_Unit_3 presente sul desktop della macchina virtuale dedicata.

Tool utilizzati per la raccolta delle informazioni:

- IDA Pro
- CFF Explorer

Per prima cosa andiamo ad aprire il tool IDA Pro presente nella nostra macchina virtuale. Una volta all'interno del programma, procediamo selezionando il Malware da analizzare nell'icona in alto a sinistra.



Per analizzare quali parametri e variabili siano passati alla funzione main occorre cliccare nella sezione “Functions” e cercare “main”.

Function name	Segment	Start	Length	R	F	L	S	B	T	=
sub_401000	.text	00401000	0000007F	R	.	.	.	B	T	.
sub_401080	.text	00401080	00000145	R	.	.	.	B	T	.
main	.text	004011D0	000000C9	R	.	.	.	B	T	.
sub_401299	.text	00401299	00000031	R
_fclose	.text	004012CA	00000056	R	.	L	.	.	T	.
_fwrite	.text	00401320	0000010A	R	.	L	.	B	T	.
_fopen	.text	0040142A	00000020	R	.	L	.	.	T	.
_fopen	.text	0040144A	00000013	R	.	L	.	.	T	.
_strchr	.text	00401460	00000027	R	.	L	.	B	T	.
start	.text	00401487	000000D4	R	.	L	.	B	.	.
_amsg_exit	.text	00401566	00000025	R	.	L	.	.	T	.
_fast_error_exit	.text	0040158B	00000024	R	.	L	S	.	T	.
_stbuf	.text	004015AF	0000008D	R	.	L
_ftbuf	.text	0040163C	0000003D	R	.	L
sub_401679	.text	00401679	0000007E	R	.	.	.	B	T	.
_write_char	.text	00401E17	00000035	R	.	L	S	B	T	.
_write_multi_char	.text	00401F4C	00000031	R	.	L	S	.	T	.

Fatto ciò otterremo un diagramma di flusso il quale ci darà le seguenti informazioni:

- Parametri passati alla funzione main: 3
- Variabili dichiarate all’interno della funzione main: 4

```

.text:004011D0
.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_8 = dword ptr -8
.text:004011D0 var_4 = dword ptr -4
.text:004011D0 argc = dword ptr 8
.text:004011D0 argv = dword ptr 0Ch
.text:004011D0 envp = dword ptr 10h
.text:004011D0

```

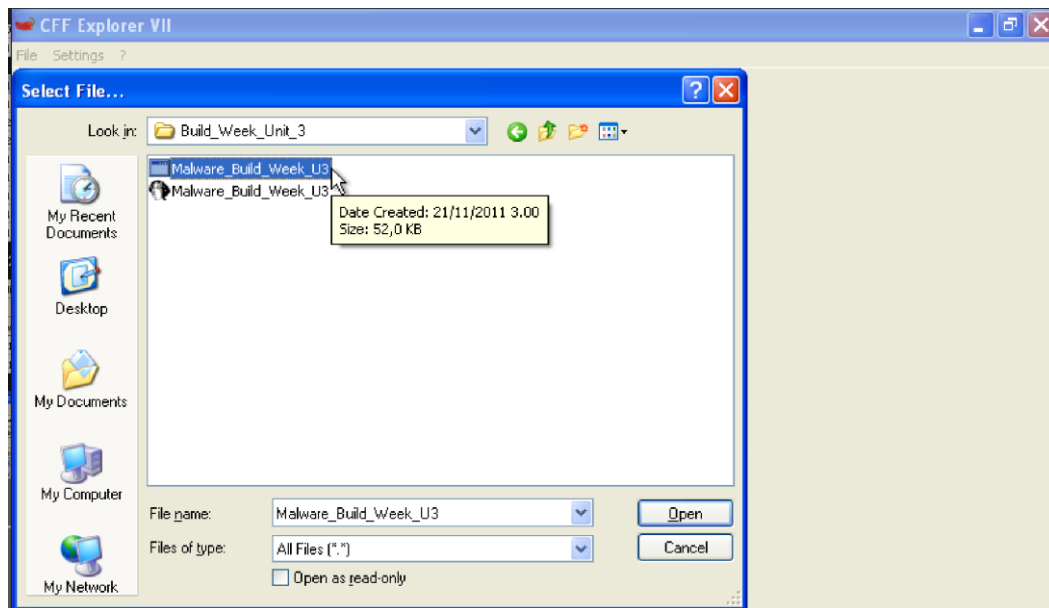
← VARIABILI

← PARAMETRI

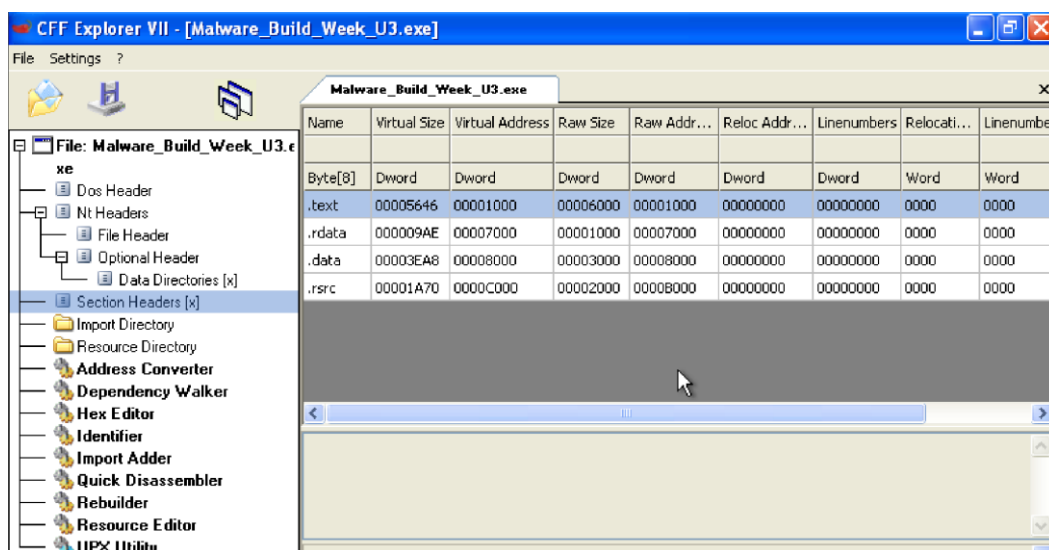
2. Sezioni presenti all'interno del file eseguibile

Per sapere quante librerie questo Malware importa potremmo sempre avvalerci di IDA Pro, o alternativamente come in questo caso, per l’analisi statica possiamo utilizzare un altro tool utilissimo: **CFF Explorer**.

Andiamo ad aprire il file contenente il Malware da analizzare.



Per cercare le sezioni contenute nel file andiamo a cliccare su “Section Headers”.



Come possiamo vedere dalle immagini questo file contiene sezioni di tipo:

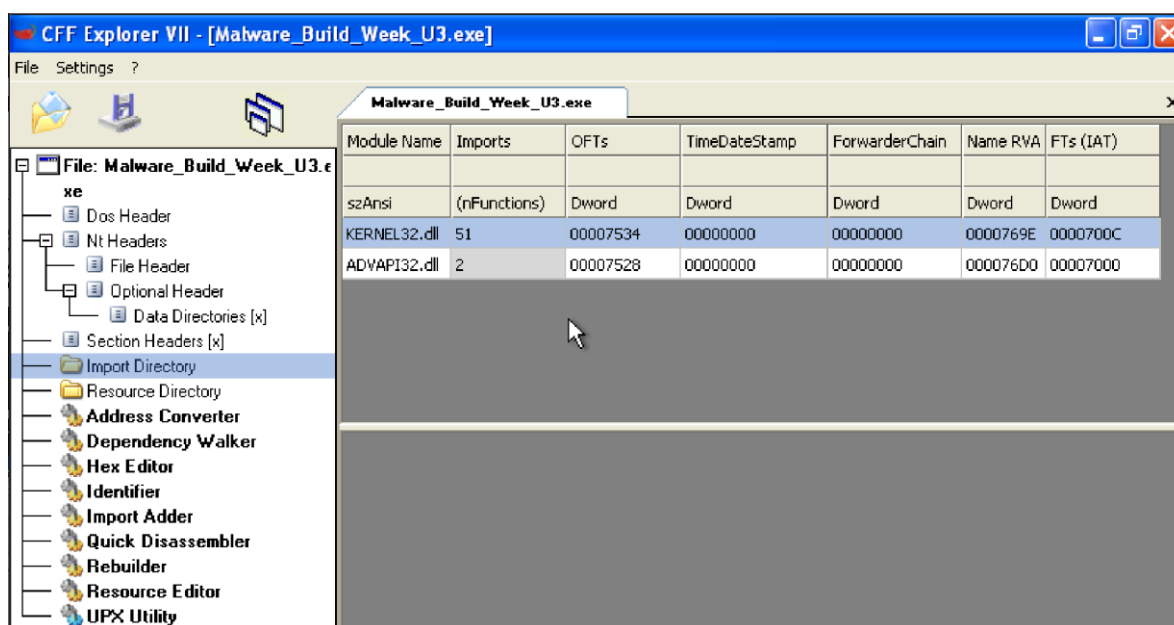
- **.text**: contiene le istruzioni che la CPU eseguirà una volta che il software sarà avviato.
- **.rdata**: include informazioni riguardanti librerie e funzioni importate ed esportate dall'eseguibile.
- **.data**: contiene dati e variabili globali del programma eseguibile, i quali devono essere disponibili in qualsiasi parte del programma.
- **.rsrc**: include le risorse utilizzate dall'eseguibile, come ad esempio icone, immagini e stringhe che non sono parte dell'eseguibile stesso

3. Librerie importate dal malware

Per quanto riguarda la ricerca delle librerie, queste si possono trovare nella sezione “**Import Directory**”.

Come possiamo vedere dall’immagine, questo malware importa due tipi di librerie:

- **KERNEL32.dll**: libreria che contiene le funzioni principali per interagire con il sistema operativo, come ad esempio manipolare i file o gestione della memoria.
- **ADVAPI32.dll**: libreria che contiene le funzioni per interagire con i servizi ed i registri del sistema operativo Microsoft



Qui sotto sono riportate le varie funzioni contenute all’interno di ciascuna libreria:

Libreria KERNEL32.dll (51 funzioni):

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
000076FE	000076FE	007D	ExitProcess
0000770C	0000770C	019F	HeapFree
00007718	00007718	011A	GetLastError

OFTs	FTs (IAT)	Hint	Name
0000756C	00007044	00007728	0000772A
Dword	Dword	Word	szAnsi
00007728	00007728	02DF	WriteFile
00007734	00007734	029E	TerminateProcess
00007748	00007748	00F7	GetCurrentProcess
0000775C	0000775C	02AD	UnhandledExceptionFilter
00007778	00007778	00B2	FreeEnvironmentStringsA
00007792	00007792	00B3	FreeEnvironmentStringsW
000077AC	000077AC	02D2	WideCharToMultiByte
000077C2	000077C2	0106	GetEnvironmentStrings
000077DA	000077DA	0108	GetEnvironmentStringsW
000077F4	000077F4	026D	SetHandleCount
00007806	00007806	0152	GetStdHandle
00007816	00007816	0115	GetFileType
00007824	00007824	0150	GetStartupInfoA
00007836	00007836	0109	GetEnvironmentVariableA

OFTs	FTs (IAT)	Hint	Name
000075A4	0000707C	00007850	00007852
Dword	Dword	Word	szAnsi
00007850	00007850	0175	GetVersionExA
00007860	00007860	019D	HeapDestroy
0000786E	0000786E	019B	HeapCreate
0000787C	0000787C	02BF	VirtualFree
0000788A	0000788A	022F	RtlUnwind
00007896	00007896	0199	HeapAlloc
000078A2	000078A2	01A2	HeapReAlloc
000078B0	000078B0	027C	SetStdHandle
000078C0	000078C0	00AA	FlushFileBuffers
000078D4	000078D4	026A	SetFilePointer
000078E6	000078E6	0034	CreateFileA
000078F4	000078F4	00BF	GetCPInfo
00007900	00007900	00B9	GetACP
0000790A	0000790A	0131	GetOEMCP

00007916	00007916	013E	GetProcAddress
00007928	00007928	01C2	LoadLibraryA
00007938	00007938	0261	SetEndOfFile
00007948	00007948	0218	ReadFile
00007954	00007954	01E4	MultiByteToWideChar
0000796A	0000796A	01BF	LCMapStringA
0000797A	0000797A	01C0	LCMapStringW
0000798A	0000798A	0153	GetStringTypeA
0000799C	0000799C	0156	GetStringTypeW

Libreria ADVAPI32.dll (2 funzioni):

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
000076AC	000076AC	0186	RegSetValueExA
000076BE	000076BE	015F	RegCreateKeyExA

Conclusioni finali:

A valle di questa analisi statica, non possiamo avere prove concrete riguardo l'identità di questo Malware.

Però possiamo ipotizzare che questo file richiamerà funzioni di modifica alle chiavi di registro tramite "RegCreateKeyExA" – "RegSetValueExA" contenute nella libreria "ADVAPI32.dll" per poi andare a creare un file ed eseguirlo con comando "CreateFileA" – "SetEndOfFile" attraverso la libreria KERNEL32.dll

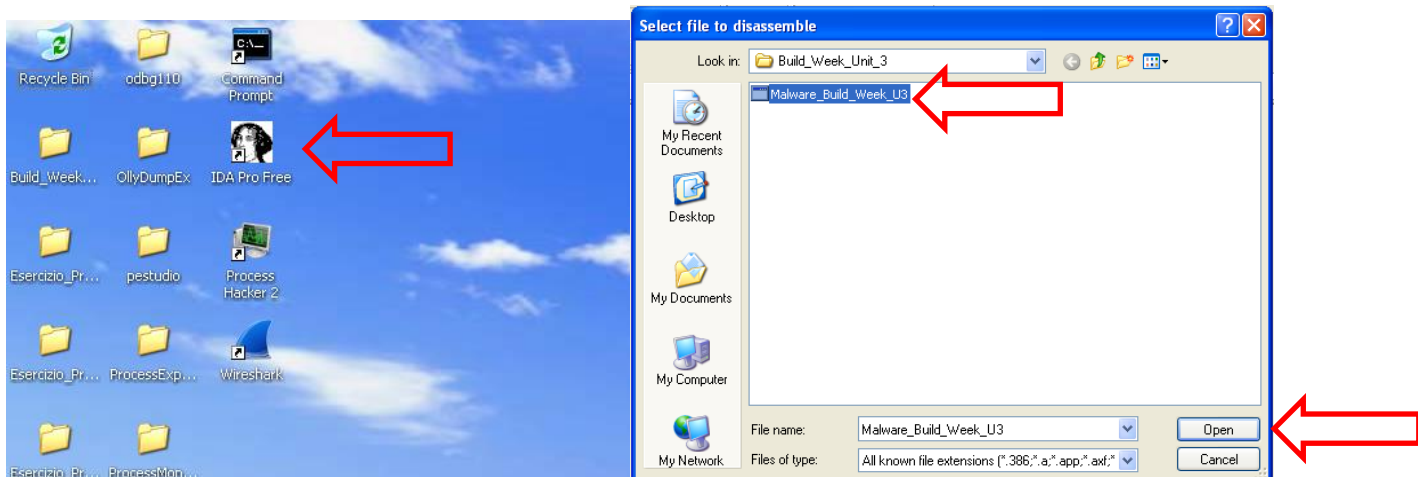
GIORNO 2

TASKS:

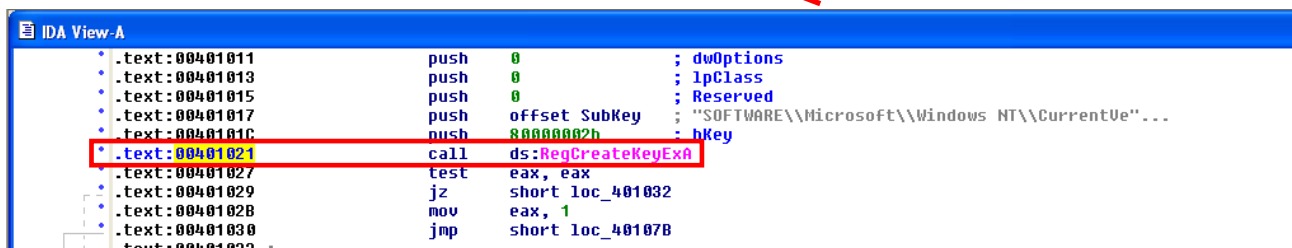
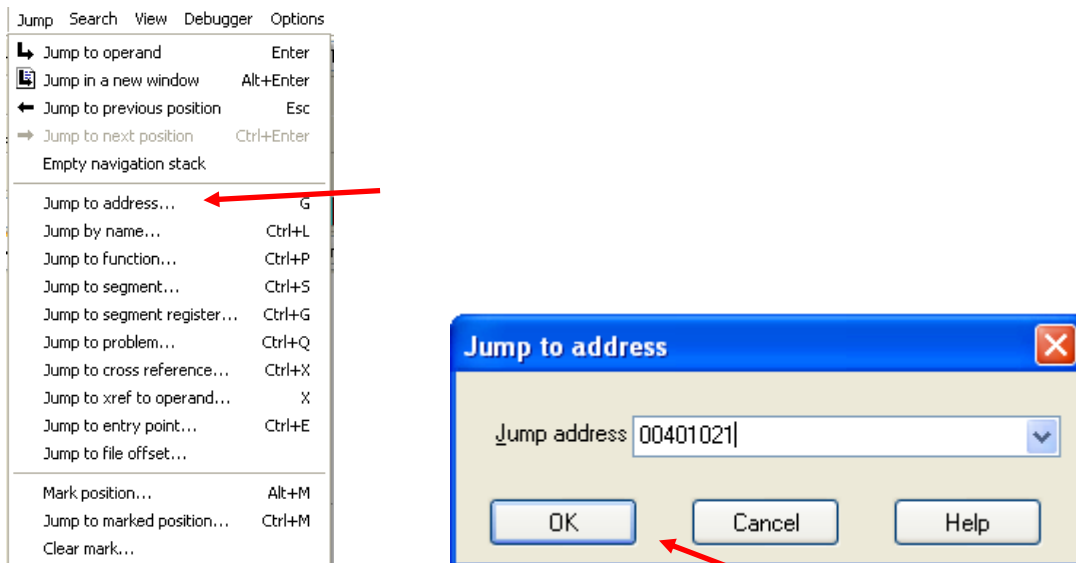
1. Lo scopo della funzione chiamata alla locazione di memoria 00401021
2. Come vengono passati i parametri alla funzione alla locazione 00401021
3. Che oggetto rappresenta il parametro alla locazione 00401017
4. Significato delle istruzioni comprese tra gli indirizzi 00401027 e 00401029
5. Con riferimento all'ultimo quesito, tradurre il codice assembly nel corrispondente costrutto C
6. Valutare ora la chiamata alla locazione 00401047, qual è il valore del parametro "ValueName"
7. Spiegare le funzionalità implementate dal malware in questa sezione

1. SCOPO DELLA FUNZIONE CREATEKEY

Per iniziare con l'analisi del malware, avvio il file eseguibile con il tool IDA presente sulla nostra macchina virtuale



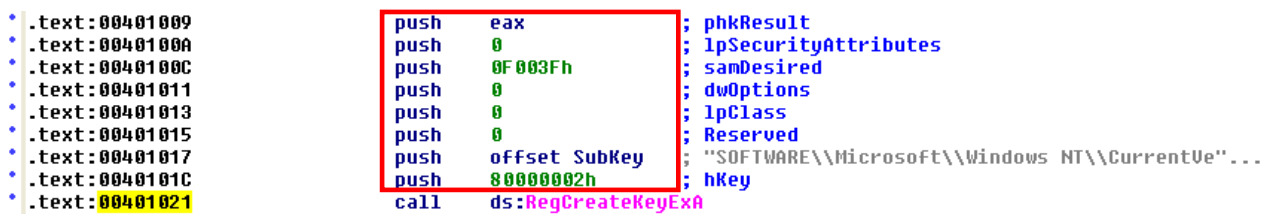
Con la funzione disassembler del tool, questo ci restituirà dall'eseguibile il file assembly. Come richiesto dalla traccia "jumperemo" all'indirizzo di memoria "00401021"



La funzione richiesta dal task serve a creare la chiave di registro specificata. Se la chiave esiste già, la funzione la apre. Si noti che i nomi delle chiavi non fanno distinzione tra maiuscole e minuscole

2. COME VENGONO PASSATI I PARAMETRI

Con le istruzioni **push** i parametri vengono passati nello stack di memoria, che la funzione poi andrà ad utilizzare



3. COSA RAPPRESENTA L'OGGETTO ALL'INDIRIZZO 00401017

L'oggetto alla locazione di memoria **00401017** rappresenta la chiave di registro utilizzata dal malware per la persistenza sul pc vittima. Questa chiave che ha come percorso "Software\\Microsoft\\Windows NT\\CurrentVersion\\WinLogon" viene creata dalla funzione "RegCreateKeyExA"


```

* .text:00401009      push     eax          ; phkResult
* .text:0040100A      push     0             ; lpSecurityAttributes
* .text:0040100C      push     0F003Fh        ; samDesired
* .text:00401011      push     0             ; dwOptions
* .text:00401013      push     0             ; lpClass
* .text:00401015      push     0             ; Reserved
* .text:00401017      push     offset SubKey ; "SOFTWARE\\Microsoft\\Windows NT\\CurrentVe"...
* .text:0040101C      push     80000002h    ; hKey
* .text:00401021      call     ds:RegCreateKeyExA

```

4. SPIEGARE LE DUE ISTRUZIONI ALL'INDIRIZZO 00401027 – 00401029

La prima istruzione è l'istruzione test nella locazione di memoria **00401027**. È un'istruzione condizionale che è simile ad un "AND" logico, ma rispetto a quest'ultimo non modifica gli operandi. Quello che modifica è la **ZF** (zero-flag) che verrà settata a 1 se e solo se il risultato dell'operazione sarà 0.

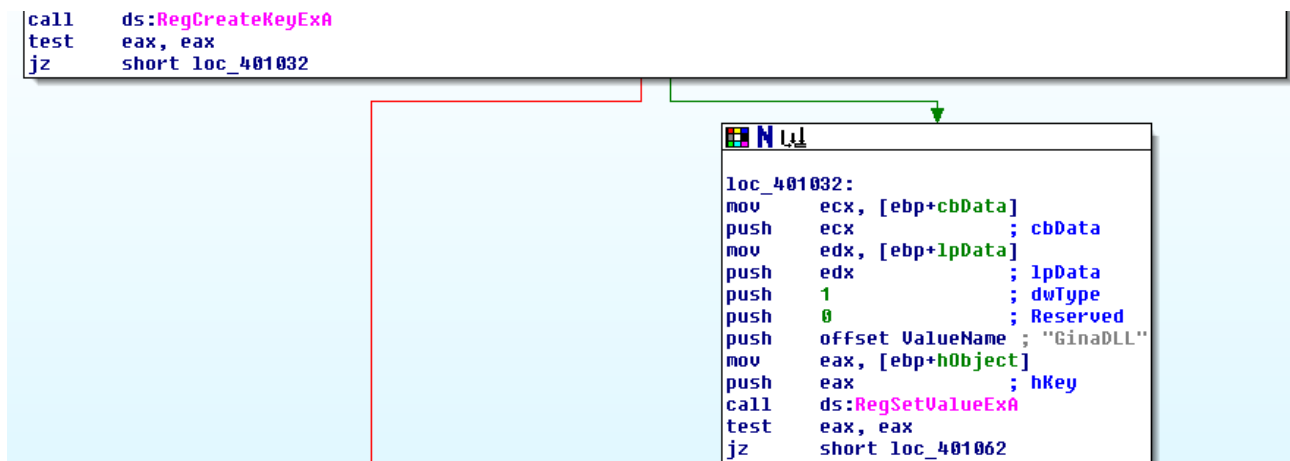
La seconda istruzione è l'istruzione "**JZ**" la quale effettuerà un salto condizionale se e solo se lo ZF dell'operazione precedente sia settato a 1. in questo caso il salto verrà effettuato verso la locazione di memoria **401032**.

```

* .text:00401027      test     eax, eax
* .text:00401029      jz       short loc_401032

```

Questo è il salto visto dal diagramma di flusso:

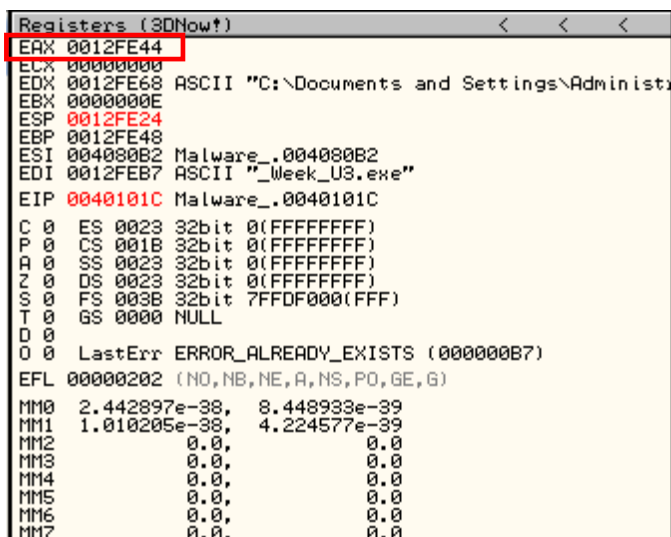


5. TRADURRE IL CODICE ASSEMBLY IN C

Con questo codice abbiamo simulato il funzionamento del costrutto IF del nostro codice assembly. Abbiamo dichiarato le variabili per fare l'operazione di test tra i due operandi, dopodiché abbiamo impostato delle condizioni in base al risultato dell'operazione. Se il risultato del test fosse 0 allora la ZF verrà impostata a 1 e il salto avverrà, diversamente si continuerà con il codice.

```
1  #include <stdio.h>
2
3  int main()
4  {
5
6      int zf; //questa variabile simula la ZF
7      int eax = 12; //questa variabile simula il registro eax
8      int test; //questa simula l'operatore test
9      test = eax - eax; // AND logico
10     if(test == 0){ // inizio del ciclo IF
11         zf = 1;
12     }
13
14     if(zf == 1){ // se la ZF sarà uguale a 1 farà il salto alla locazione "401032"
15         goto loc_401032;
16     }else{
17         goto loc_40107B; // altrimenti continuerà con il codice
18     }
19
20     loc_401032: printf("Salto avvenuto");
21     loc_40107B: printf("Salto non avvenuto");
22     return 0;
23 }
```

Il valore che abbiamo impostato al registro "EAX" è 12, poiché è il valore decimale della versione esadecimale (0012FE44) che siamo riusciti a trovare tramite l'utilizzo di OllyDBG



The screenshot shows the 'Registers (3DNow!)' window in OllyDBG. The EAX register is highlighted with a red box and contains the value 0012FE44. Other registers and their values are listed below:

Register	Value
EAX	0012FE44
ECX	00000000
EDX	0012FE68 ASCII "C:\Documents and Settings\Administr...
EBX	0000000E
ESP	0012FE24
EBP	0012FE48
ESI	004080B2 Malware_.004080B2
EDI	0012FEB7 ASCII "_Week_US.exe"
EIP	0040101C Malware_.0040101C
C 0	ES 0023 32bit 0(FFFFFFFF)
P 0	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDF000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_ALREADY_EXISTS (000000B7)
EFL	00000202 (NO,NB,NE,A,NS,PO,GE,G)
MM0	2.442897e-38, 8.448933e-39
MM1	1.010205e-38, 4.224577e-39
MM2	0.0, 0.0
MM3	0.0, 0.0
MM4	0.0, 0.0
MM5	0.0, 0.0
MM6	0.0, 0.0
MM7	0.0, 0.0

6. VALORE DEL PARAMETRO "VALUEName"

Con riferimento alla chiamata di funzione nella locazione di memoria **00401047**, il valore che viene passato al parametro "ValueName" è **GinaDLL**

```

.text:00401032      mov     ecx, [ebp+cbData]
.text:00401035      push    ecx                ; cbData
.text:00401036      mov     edx, [ebp+lpData]
.text:00401039      push    edx                ; lpData
.text:0040103A      push    1                  ; dwType
.text:0040103C      push    0                  ; Reserved
.text:0040103E      push    offset ValueName ; "GinaDLL"
.text:00401043      mov     eax, [ebp+hObject]
.text:00401046      push    eax                ; hKey
.text:00401047      call    ds:RegSetValueExA

```

Per completezza abbiamo lanciato il file eseguibile del malware con il tool “OllyDBG” per verificare che il valore di “ValueName” fosse effettivamente quello riscontrato con IDA

00401035	51	PUSH ECX	BufSize
00401036	8B55 08	MOV EDX,DWORD PTR SS:[EBP+8]	Buffer
00401039	52	PUSH EDX	ValueType = REG_SZ
0040103A	6A 01	PUSH 1	Reserved = 0
0040103C	6A 00	PUSH 0	ValueName = "GinaDLL"
0040103E	68 4C804000	PUSH Malware_.0040804C	hKey
00401043	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]	RegSetValueExA
00401046	50	PUSH EAX	
00401047	FF15 00704000	CALL DWORD PTR DS:[<&ADVAPI32.RegSetVal	

7. FUNZIONALITA' IMPLEMENTATE NELLA SEZIONE DI CODICE ANALIZZATA

Con le righe di codice a nostra disposizione siamo riusciti a ipotizzare che il malware cerca di ottenere la persistenza all'interno del registro di Windows. Per fare ciò si serve della funzione “RegCreateKeyExA” (che crea la chiave di registro) e della funzione “RegSetValueExA” (che la configura).

GIORNO 3

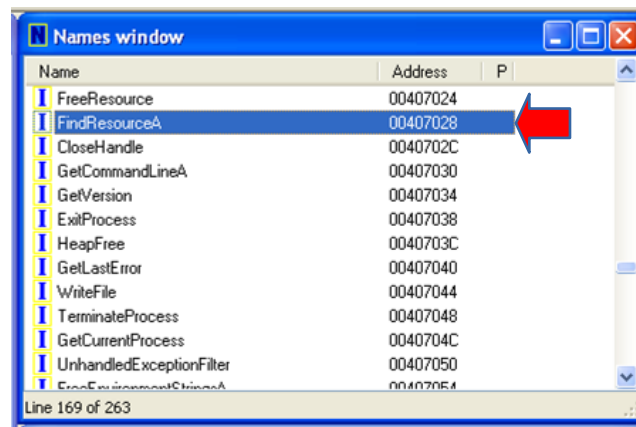
TASKS:

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

1. Mostrare il valore del parametro ResourceName passato alla funzione FindResourceA()
2. Date le chiamate di funzione che il Malware effettua in questa sezione di codice, analizzare le funzionalità che sta implementando
3. Analizzare se è possibile identificare questa funzionalità utilizzando l'analisi statica basica
4. In caso di risposta affermativa al punto precedente, elencare le evidenze a supporto
5. Disegnare un diagramma di flusso che comprenda le funzioni

1) Valore del parametro ResourceName passato alla funzione FindResourceA()

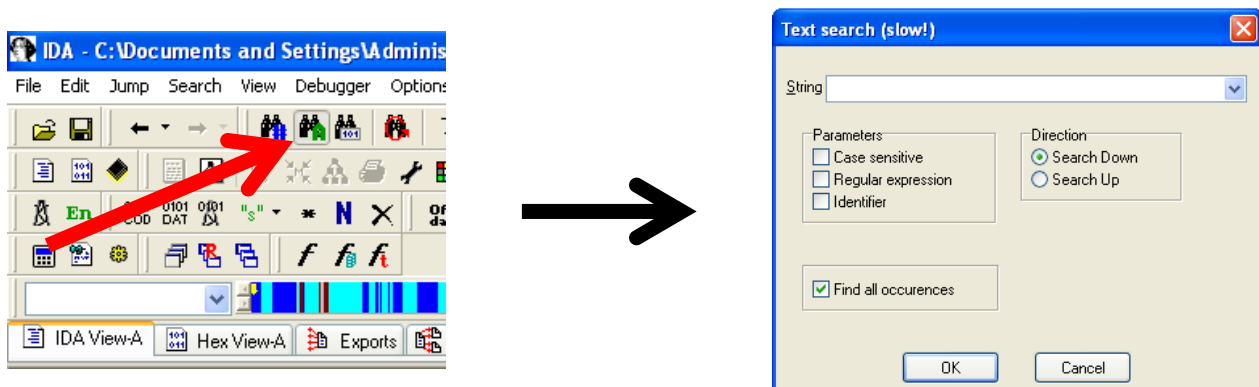
Per prima cosa siamo andati ad avviare il tool IDA Pro. Una volta selezionato il malware che deve essere analizzato abbiamo cercato nella finestra “Names” il nome della funzione che ci interessa: **FindResourceA()**:



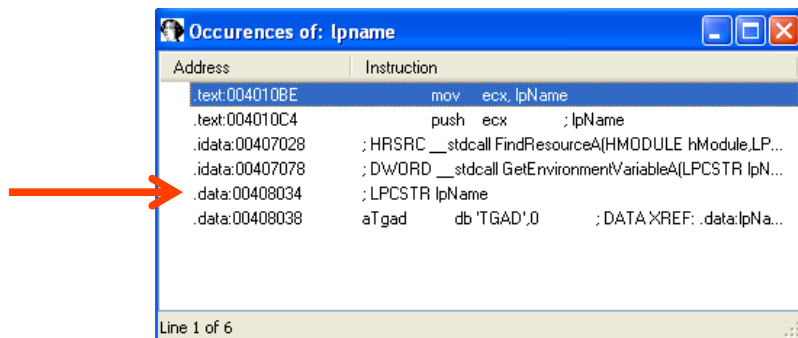
Facendo doppio click con il mouse su questa funzione siamo andati alla sua porzione di codice:

```
text:004010B8
text:004010B8 loc_4010B8:
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
```

Tuttavia, con questa vista non siamo riusciti a vedere nella porzione di codice mostrato nell’immagine precedente il valore del parametro “ResourceName”. Siamo quindi andati ad eseguire una ricerca all’interno del codice malware tramite l’apposito strumento “search”.



All’interno del campo “string” siamo andati a inserire il valore lpName. Questa ricerca ci ha fornito i seguenti risultati:



Scorrendo tra i vari risultati ottenuti dalla nostra ricerca siamo infine giunti alla seguente porzione di codice, corrispondente al risultato evidenziato dalla freccia rossa nell'immagine qui sopra:

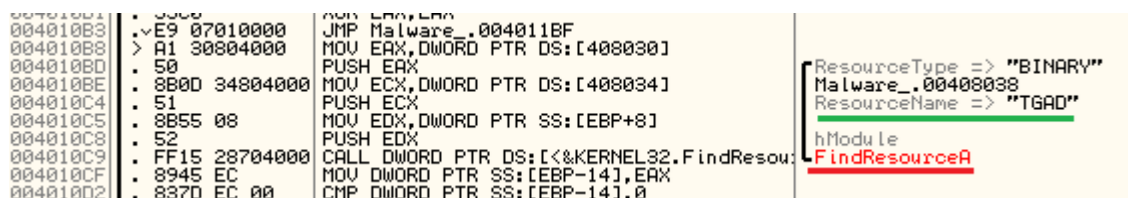
```
.data:00408030 ; LPCSTR lpType
.data:00408030 lpType      dd offset aBinary      ; DATA XREF: sub_401080:loc_401088↑r
.data:00408030 ; "BINARY"
.data:00408034 ; LPCSTR lpName
.data:00408034 lpName      dd offset aTgad      ; DATA XREF: sub_401080+3E↑r
.data:00408034 ; "TGAD"
.data:00408038 aTgad      db 'TGAD',0      ; DATA XREF: .data:lpName↑o
.data:0040803D align 10h
.data:00408040 aBinary     db 'BINARY',0      ; DATA XREF: .data:lpType↑o
.data:00408047 align 4
.data:00408048 aRi        db 'RI',0Ah,0      ; DATA XREF: sub_401080:loc_401062↑o
```

Dall'immagine qui sopra possiamo vedere come il valore del parametro lpName, che viene dato alla funzione FindResourceA() è "TGAD". XREF è una feature per mostrare da dove sono state chiamate determinate funzioni e oggetti o quali funzioni e oggetti sono usati da una funzione specifica. Nel nostro caso possiamo vedere come XREF ci mostra che il parametro lpName viene utilizzato dalla subroutine che inizia alla locazione di memoria 00401080.

Verifica su OllyDBG

Per avere conferma che la nostra ricerca fosse corretta, siamo andati ad analizzare il malware con il tool OllyDBG, che fornisce tali informazioni in maniera più immediata.

Infatti, spostandoci alla locazione di memoria 004010C9, dove è situata la funzione che stiamo analizzando, abbiamo questa schermata:



Da essa possiamo vedere chiaramente, sottolineato in verde, che il parametro "ResourceName" passato alla funzione FindResourceA() assume il valore "TGAD".

2) Date le chiamate di funzione che il Malware effettua in questa sezione di codice, analizzare le funzionalità che sta implementando

Nella sezione di codice che dobbiamo analizzare per questo task abbiamo trovato 4 funzioni che vengono richiamate tramite l'istruzione "call":

FindResourceA

```
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      -----
```

Questa funzione determina la posizione di una risorsa con il tipo e il nome specificati nel modulo. Come abbiamo visto al punto precedente la risorsa che di cui viene determinata la posizione è il TGAD.

LoadResource

```
.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
```

Questa funzione recupera un handle che può essere usato per ottenere un puntatore al primo byte della risorsa specificata in memoria. Tramite questa funzione viene quindi recuperato l'handle per la risorsa determinata con la funzione precedente.

LockResource

```
.text:004010FB
.text:004010FB loc_4010FB:                ; CODE XREF: sub_401080+74↑j
.text:004010FB      mov     edx, [ebp+hResData]
.text:004010FE      push    edx                ; hResData
.text:004010FF      call    ds:LockResource
.text:00401105      mov     [ebp+var_8], eax
.text:00401108      cmp     [ebp+var_8], 0
.text:0040110C      jnz     short loc_401113
.text:0040110E      jmp     loc_4011A5
```

Questa funzione recupera un puntatore alla risorsa specificata in memoria, quindi ci fornisce un puntatore al primo byte della risorsa TGAD.

SizeofResource

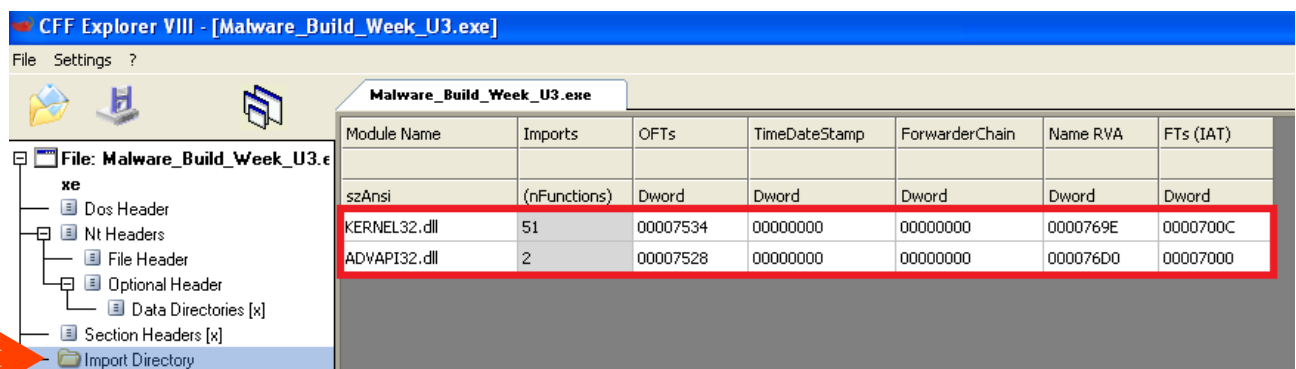
```
.text:00401113  
.text:00401113 loc_401113: ; CODE XREF: sub_401080+8C↑j  
.text:00401113 mov     eax, [ebp+hResInfo]  
.text:00401116 push    eax ; hResInfo  
.text:00401117 mov     ecx, [ebp+hModule]  
.text:0040111A push    ecx ; hModule  
.text:0040111B call    ds:SizeofResource  
.text:00401121 mov     [ebp+dwSize], eax  
.text:00401124 cmp     [ebp+dwSize], 0  
.text:00401128 ja      short loc_40112C  
.text:0040112A jmp     short loc_4011A5
```

Questa funzione recupera le dimensioni, in byte, della risorsa specificata; tramite questa funzione andiamo quindi a vedere le dimensioni in byte della risorsa TGAD.

Dopo avere analizzato le chiamate di funzioni inserite nella porzione di codice da analizzare, possiamo andare ad ipotizzare che il malware che stiamo analizzando sia un **dropper**. I dropper sono dei malware creati per installare un malware, un virus, o aprire una backdoor su un sistema.

3) Analizzare se è possibile identificare questa funzionalità utilizzando l'analisi statica basica

Andando ad eseguire l'analisi statica basica con il tool [CFF Explorer](#) possiamo andare ad esaminare le funzioni e le librerie che vengono importate dal malware, come possiamo vedere nell'immagine qui sotto:



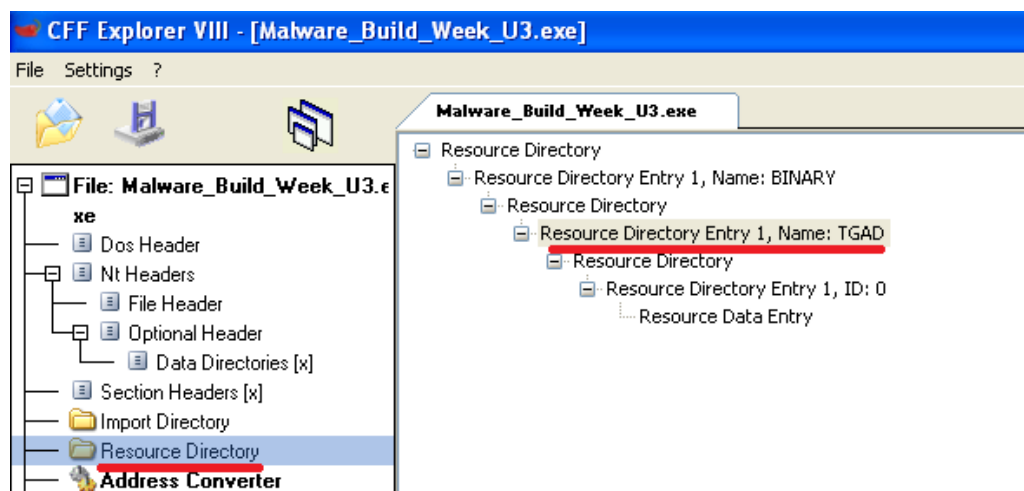
Infatti spostandoci alla voce **“Import Directory”** possiamo vedere che il malware va ad importare dalla libreria KERNEL32.dll 51 funzioni che sono le seguenti:

OFTs	FTs (IAT)	Hint	Name	OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi	Dword	Dword	Word	szAnsi
00007632	00007632	0295	SizeOfResource	00007824	00007824	0150	GetStartupInfoA
00007644	00007644	01D5	LockResource	00007836	00007836	0109	GetEnvironmentVariableA
00007654	00007654	01C7	LoadResource	00007850	00007850	0175	GetVersionExA
00007622	00007622	02BB	VirtualAlloc	00007860	00007860	019D	HeapDestroy
00007674	00007674	0124	GetModuleFileNameA	0000786E	0000786E	019B	HeapCreate
0000768A	0000768A	0126	GetModuleHandleA	0000787C	0000787C	02BF	VirtualFree
00007612	00007612	00B6	FreeResource	0000788A	0000788A	022F	RtlUnwind
00007664	00007664	00A3	FindResourceA	00007896	00007896	0199	HeapAlloc
00007604	00007604	001B	CloseHandle	000078A2	000078A2	01A2	HeapReAlloc
000076DE	000076DE	00CA	GetCommandLineA	000078B0	000078B0	027C	SetStdHandle
000076F0	000076F0	0174	GetVersion	000078C0	000078C0	00AA	FlushFileBuffers
000076FE	000076FE	007D	ExitProcess	000078D4	000078D4	026A	SetFilePointer
0000770C	0000770C	019F	HeapFree	000078E6	000078E6	0034	CreateFileA
00007718	00007718	011A	GetLastError	000078F4	000078F4	00BF	GetCPInfo
00007728	00007728	02DF	WriteFile	00007900	00007900	00B9	GetACP
00007734	00007734	029E	TerminateProcess	0000790A	0000790A	0131	GetOEMCP
00007748	00007748	00F7	GetCurrentProcess	00007916	00007916	013E	GetProcAddress
0000775C	0000775C	02AD	UnhandledExceptionFilter	00007928	00007928	01C2	LoadLibraryA
00007778	00007778	00B2	FreeEnvironmentStringsA	00007938	00007938	0261	SetEndOfFile
00007792	00007792	00B3	FreeEnvironmentStringsW	00007948	00007948	0218	ReadFile
000077AC	000077AC	02D2	WideCharToMultiByte	00007954	00007954	01E4	MultiByteToWideChar
000077C2	000077C2	0106	GetEnvironmentStrings	0000796A	0000796A	01BF	LCMapStringA
000077DA	000077DA	0108	GetEnvironmentStringsW	0000797A	0000797A	01C0	LCMapStringW
000077F4	000077F4	026D	SetHandleCount	0000798A	0000798A	0153	GetStringTypeA
00007806	00007806	0152	GetStdHandle	0000799C	0000799C	0156	GetStringTypeW
00007816	00007816	0115	GetFileType				

Come possiamo vedere dall'immagine qui sopra, sottolineate in rosso troviamo le funzioni che abbiamo analizzato al punto precedente.

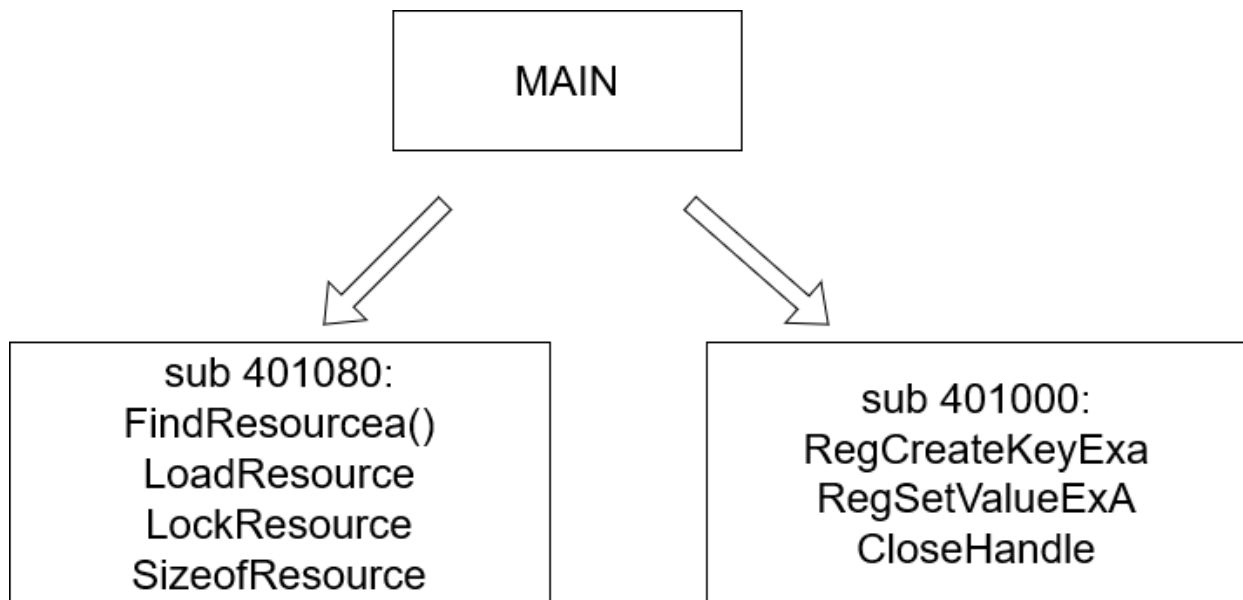
4) In caso di risposta affermativa al punto precedente, elencare le evidenze a supporto

Come abbiamo visto al punto precedente, tramite l'utilizzo del tool CFF Explorer è possibile andare a vedere quali sono le funzioni importate dalle librerie. In aggiunta a ciò, spostandoci sulla sezione **Resource Directory**, possiamo vedere come sia elencata anche la risorsa richiamata dalla funzione FindResourceA(), come mostrato dall'immagine qui sotto:



Tuttavia è consigliabile eseguire l'analisi statica avanzata e l'analisi dinamica per avere una migliore comprensione delle attività svolte dal malware.

5) Disegnare un diagramma di flusso che comprenda le funzioni analizzate



GIORNO 4

ANALISI DINAMICA BASICA

Analisi del malware *Malware_Build_Week_U3.exe*

Tasks:

1. Preparazione dell'ambiente di test sulla VM ed esecuzione del malware
2. Rilevazione di eventuali modifiche all'interno della cartella di origine dell'eseguibile in seguito all'esecuzione del malware
3. Analisi dell'interazione del malware con il **registro**: identificazione della chiave di registro creata e del valore ad essa associato
4. Analisi dell'interazione del malware con il **file system**: identificazione della chiamata di sistema responsabile della modifica del contenuto della cartella di origine dell'eseguibile

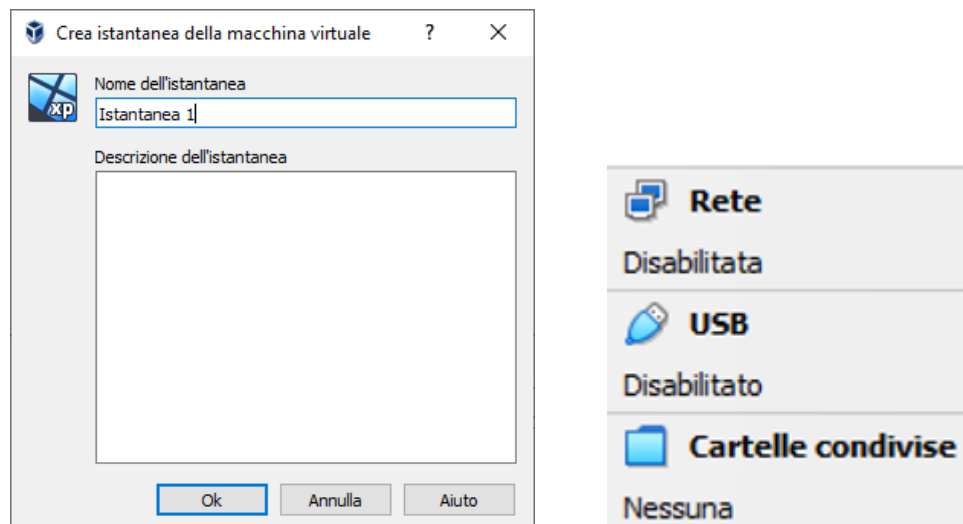
5. Descrizione del comportamento del malware in base alle informazioni raccolte tramite analisi statica e dinamica

1. Preparazione dell'ambiente di test sulla VM ed esecuzione del malware

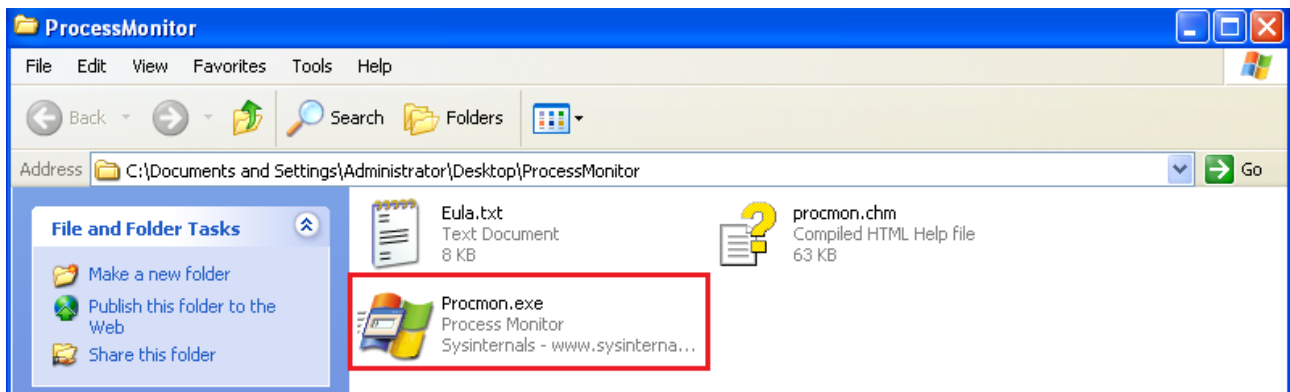
Le attività odierne sono incentrate sull'analisi dinamica basica del file eseguibile di test **Malware_Build_Week_U3.exe**.

L'**analisi dinamica basica** comprende una gamma di attività di analisi che presuppongono l'**esecuzione del malware** in ambiente dedicato e protetto. È generalmente effettuata dopo l'analisi statica basica, per sopperire ai limiti di quest'ultima (infatti, al contrario dell'analisi statica, l'analisi dinamica permette di osservare in maniera diretta le funzionalità di un malware in esecuzione su un sistema) e confermare o smentire le ipotesi formulate durante l'analisi statica in merito al funzionamento del malware.

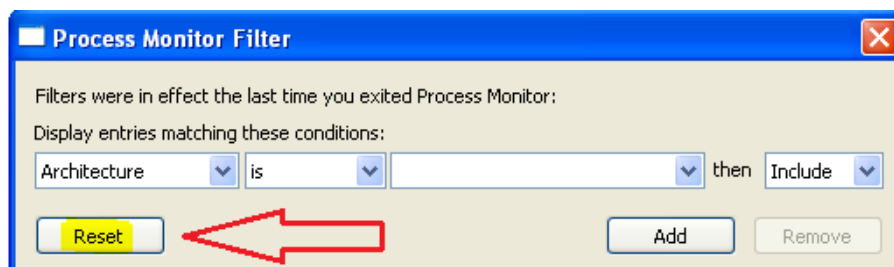
A monte delle operazioni di analisi, è utile creare un'istantanea della macchina virtuale di test (*Malware Analysis_Final* – OS Windows XP SP3) per poter eventualmente ripristinare le funzionalità della stessa in caso di problemi generati dall'esecuzione del malware; altrettanto importante è **isolare** l'ambiente di test disabilitando le schede di rete, la connettività USB ed eventuali cartelle condivise:



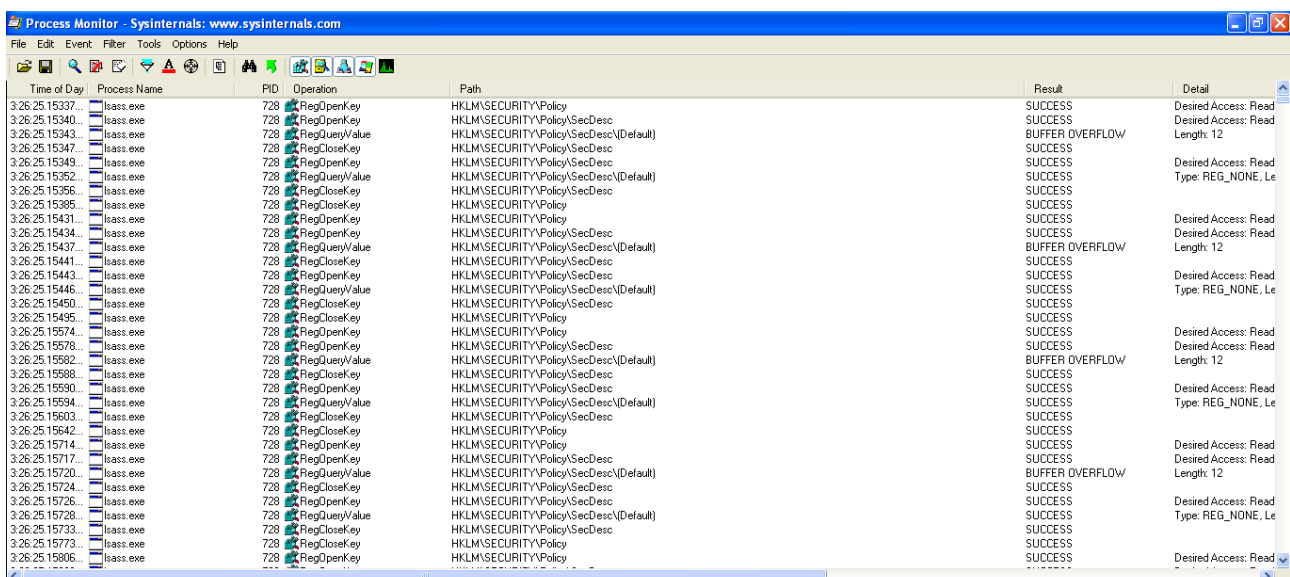
Adesso possiamo avviare la VM. Per le attività di analisi odierne, ci serviremo di **Process Monitor**: si tratta di un tool per sistemi operativi Windows che permette di monitorare i processi attivi, il file system, il registro di Windows e l'attività di rete.



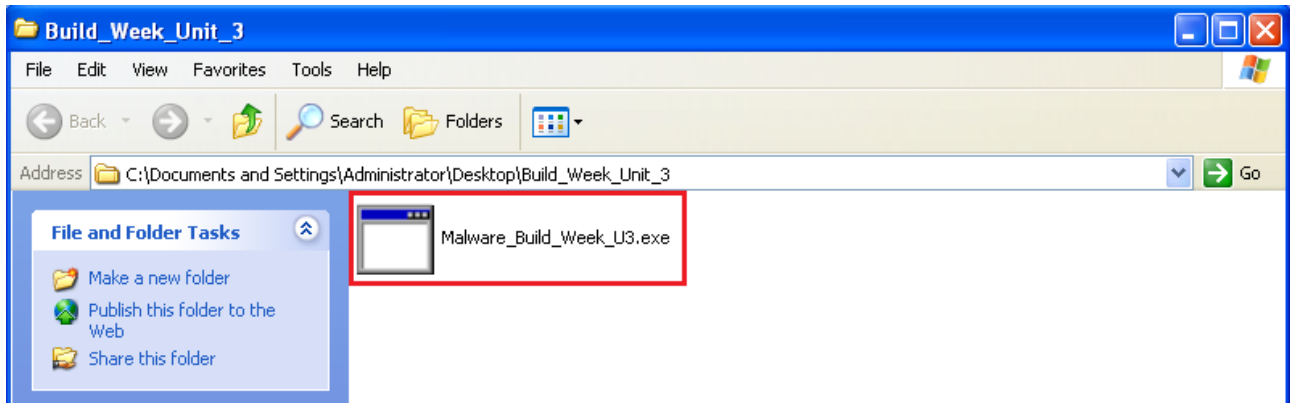
Una volta avviato il tool, ci accertiamo che non sia attivo alcun filtro cliccando sul tasto “Reset” per avere una panoramica completa della situazione iniziale:



Confermiamo la scelta con “Apply” ed “OK” ed il software inizia la cattura in tempo reale degli eventi relativi ai parametri di analisi già citati:

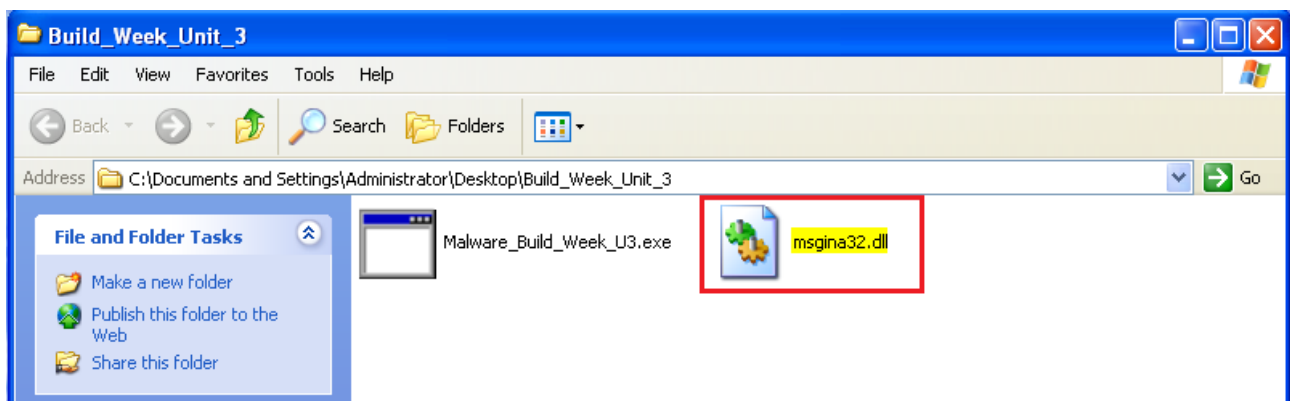


Adesso possiamo eseguire il malware da analizzare. Lo troviamo all'interno del path **C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3**. Lo avviamo facendo doppio click sull'icona dell'eseguibile.



2. Rilevazione di eventuali modifiche all'interno della cartella di origine dell'eseguibile in seguito all'esecuzione del malware

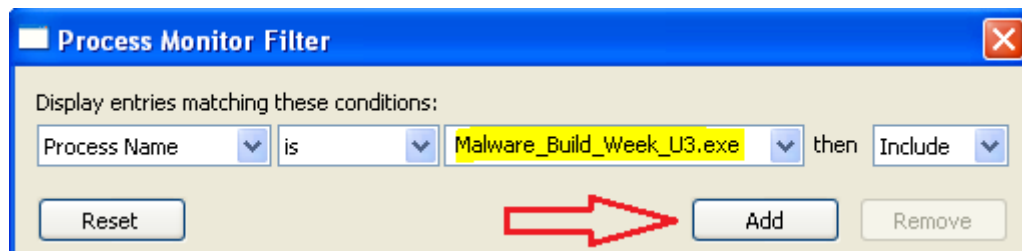
In seguito all'avvio del malware, notiamo che all'interno della cartella di origine dello stesso viene creato un nuovo file dal nome **msgina32.dll**:




Quanto appena visto conferma le ipotesi formulate durante l'analisi statica: il rilevamento delle chiamate di funzione **FindResource**, **LoadResource**, **LockResource** e **SizeofResource** importate dalla libreria **KERNEL32.dll** indica effettivamente che l'eseguibile analizzato è un **dropper**, ossia un programma malevolo che contiene al suo interno (specificamente, nella sezione "risorse" **.rsrc** dell'header del formato PE) un malware. Nel momento in cui viene eseguito, un dropper estrae il malware che contiene per avviarlo oppure salvarlo sul disco – in questo caso nello stesso path in cui si trova l'eseguibile.

3. Analisi dell'interazione del malware con il **registro**: identificazione della chiave di registro creata e del valore ad essa associato

Per analizzare le modifiche apportate dal malware al sistema, filtriamo i risultati ottenuti dalla cattura di Process Monitor nel seguente modo:

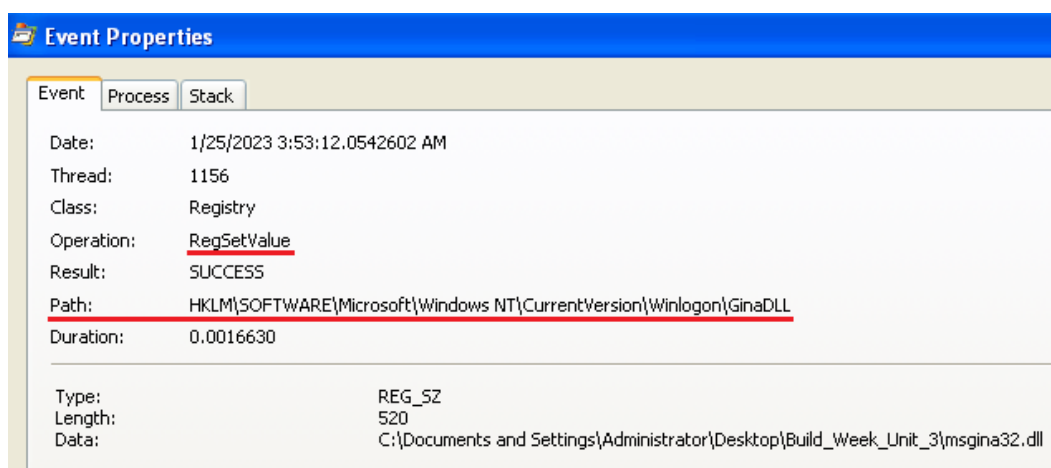


In particolare, vogliamo filtrare ulteriormente i risultati in modo da mostrare solamente gli eventi relativi all'interazione del malware con il registro di Windows. Le **chiavi di registro** sono variabili di configurazione di sistema: di conseguenza è importante, in fase di analisi, conoscere le modifiche eventualmente apportate ad esse da parte di un malware, per capire quali configurazioni sono state alterate. I valori delle chiavi rappresentano tutto ciò che viene caricato all'avvio del sistema.

Impostiamo dunque il filtro sulla sezione *Show Registry Activity* → 

Andiamo dunque ad analizzare gli eventi in base al filtro appena inserito e notiamo la seguente situazione:

Malware_Build_Week_U3...	1868	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\kernel32.dll
Malware_Build_Week_U3...	1868	RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon
Malware_Build_Week_U3...	1868	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
Malware_Build_Week_U3...	1868	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon

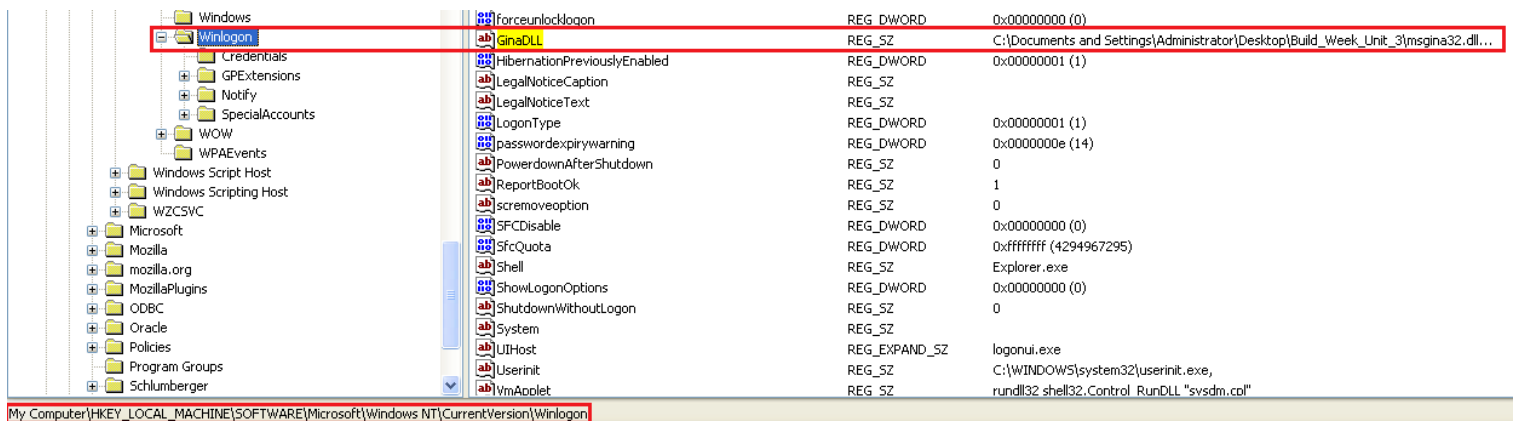


Come si può notare, tramite le chiamate di funzione **RegCreateKey** e **RegSetValue** il malware ha generato una chiave di registro all'interno del percorso **Software\Microsoft\Windows NT\CurrentVersion\Winlogon** e le ha successivamente assegnato il valore di "GinaDLL". Abbiamo verificato questa informazione anche servendoci di altri tool:

OllyDBG

<pre> 00401035 . 51 PUSH ECX 00401036 . 8B55 08 MOV EDX,DWORD PTR SS:[EBP+8] 00401039 . 52 PUSH EDX 0040103A . 6A 01 PUSH 1 0040103C . 6A 00 PUSH 0 0040103E . 68 4C804000 PUSH Malware_.0040804C 00401043 . 8B45 FC MOV EAX,DWORD PTR SS:[EBP-4] 00401046 . 50 PUSH EAX 00401047 . FF15 00704000 CALL DWORD PTR DS:[<&ADVAPI32.RegSetValueExA>] </pre>	<pre> BufSize Buffer ValueType = REG_SZ Reserved = 0 ValueName = "GinaDLL" hKey RegSetValueExA </pre>
--	---

Regedit (= *Registry Editor*, è uno strumento installato di default sui sistemi operativi Windows che viene utilizzato per visualizzare e/o modificare tutte le chiavi di registro)



4. Analisi dell'interazione del malware con il **file system**: identificazione della chiamata di sistema responsabile della modifica del contenuto della cartella di origine dell'eseguibile

Adesso vogliamo invece analizzare l'interazione del malware con il file system allo scopo di individuare la chiamata di sistema responsabile della modifica del contenuto della cartella in cui si trova l'eseguibile in esame.

A tale scopo, impostiamo il filtro sulla sezione *Show File System Activity* →

Analizzando i risultati ottenuti, possiamo notare la presenza della chiamata di funzione **CreateFile** e, poco più avanti, della chiamata di funzione **WriteFile**:

Malware_Build_Week_U3...	1868	CreateFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll
Malware_Build_Week_U3...	1868	CreateFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3
Malware_Build_Week_U3...	1868	CloseFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3
Malware_Build_Week_U3...	1868	WriteFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll

Come si vede, entrambe le chiamate di funzione puntano al percorso in cui si trova il malware ed il loro scopo è quello di creare un file vuoto (**CreateFile**) e scrivere al suo interno il malware appena estratto (**WriteFile**).

Quanto appena emerso non ci sorprende in quanto il comportamento tipico di un dropper propone, generalmente, due possibili scenari che fanno seguito all'estrazione del malware:

- La **creazione di un processo** (tramite chiamata di funzione *CreateProcess*), al fine di eseguire immediatamente il malware appena estratto
- Il **salvataggio del malware sul disco** in vista di un futuro utilizzo. In questo scenario, analogamente a quanto visto oggi, il dropper utilizzerà la coppia di funzioni *CreateFile* e *WriteFile* rispettivamente per creare un file vuoto e scrivere al suo interno il malware appena estratto

5. Descrizione del comportamento del malware in base alle informazioni raccolte tramite analisi statica e dinamica

A valle delle attività di analisi statica e dinamica fin qui svolte, è possibile trarre delle conclusioni sul comportamento del malware in esame. Inoltre, la verifica delle ipotesi formulate durante la fase di analisi statica ha avuto esito positivo: l'analisi dinamica ha confermato che il malware in oggetto

- **è un dropper**, in quanto contiene al suo interno un malware che viene effettivamente estratto (e salvato, in questo caso) al momento dell'esecuzione del file
- **ottiene la persistenza** dentro il sistema della macchina vittima, in quanto aggiunge se stesso alle entry dei programmi che devono essere eseguiti all'avvio del PC, in modo tale da essere avviato in maniera automatica e senza alcun intervento da parte dell'utente. A tale scopo, il malware crea una nuova chiave di registro tramite la chiamata di funzione *RegCreateKeyExA* e la configura con i valori desiderati tramite la chiamata di funzione *RegSetValueExA*. In fase di analisi dinamica, per verificare l'effettivo ottenimento della persistenza abbiamo riavviato la VM in seguito all'esecuzione del malware ed abbiamo potuto constatare che, al riavvio, la chiave "GinaDLL" è ancora presente nel registro di Windows.

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

Per rispondere a questa domanda è necessario ricordare il funzionamento originario del file **gina.dll**.

Il GINA opera nel contesto del processo Winlogon e come tale, la DLL GINA viene caricata nel processo di avvio del S.O. Lo scopo di una DLL GINA è quello di fornire procedure di identificazione e autenticazione dell'utente.

Dunque, il GINA predefinito esegue questa operazione delegando il monitoraggio degli eventi SAS a Winlogon. **“fonte: [GINA - Win32 apps | Microsoft Learn](#)”**

Possiamo quindi affermare che la sostituzione della dll originale con una malevola che ha come funzionalità l'intercettazione dei dati può generare una connessione remota, possibilmente in **reverse_tcp** creando una connessione dalla macchina vittima alla macchina attaccante. Possiamo dunque inserire questa tipologia di malware nella famiglia degli spyware in quanto permette di reperire i dati di accesso, mediante la generazione di un file e la scrittura di dati sensibili su di esso.

A favore della nostra tesi è stata presa in analisi una vulnerabilità del 2018 (**CVE-2018-5353**) che vede come protagonista un modulo “manomesso” di GINA.

fonte: [CVE-2018-5353 The custom GINA/CP module in Zoho ManageEngine A... \(vulmon.com\)](#)

CVE-2018-5353
Published: 30/09/2020 Updated: 15/10/2020
CVSS v2 Base Score: 7.5 | Impact Score: 6.4 | Exploitability Score: 10
CVSS v3 Base Score: 9.8 | Impact Score: 5.9 | Exploitability Score: 3.9
Vector: AV:N/AC:L/Au:N/C:P/I:P/A:P
[Subscribe to Zohocorp](#)

Vulnerability Summary
The custom GINA/CP module in Zoho ManageEngine ADService Plus prior to 5.5 build 5517 allows remote malicious users to execute code and escalate privileges via spoofing. It does not authenticate the intended server before opening a browser window. An unauthenticated attacker capable of conducting a spoofing attack can redirect the browser to gain execution in the context of the WinLogon.exe process. If Network Level Authentication is not enforced, the vulnerability can be exploited via RDP. Additionally, if the web server has a misconfigured certificate then no spoofing attack is required

Most Upvoted Vulmon Research Post
There is no Researcher post for this vulnerability
Would you like to share something about it? [Sign up](#) now to share your knowledge with the community.

Vulnerability Trend
A line chart showing the vulnerability trend over time from 2022-12-26 to 2023-01-24. The y-axis ranges from -1.0 to 1.0. The trend is relatively flat, staying near 0.0.

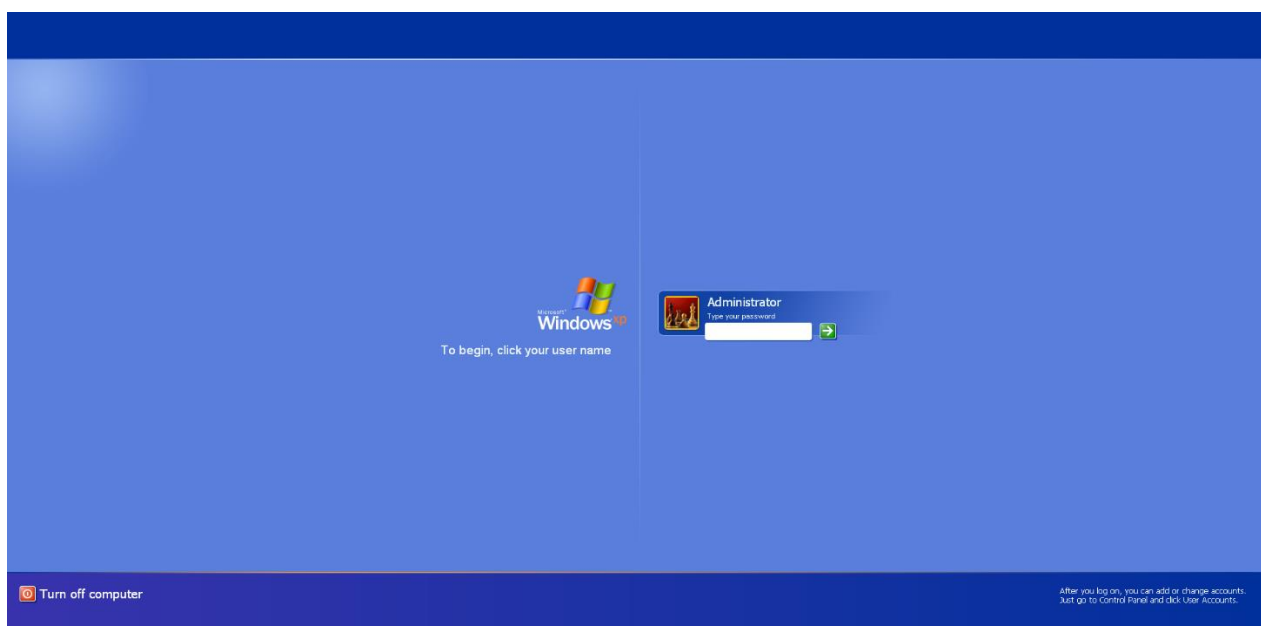
CVSSv3 - RECOMMENDATIONS:
CVE-2022-4706
CVE-2023-0447
encryption
CVE-2023-24059
cross-site request forgery
CVE-2022-42864
IDOR
CVE-2023-22483
CVE-2022-42856

Vulnerability Notification Service
You don't have to wait for vulnerability scanning results
[Get Started](#)

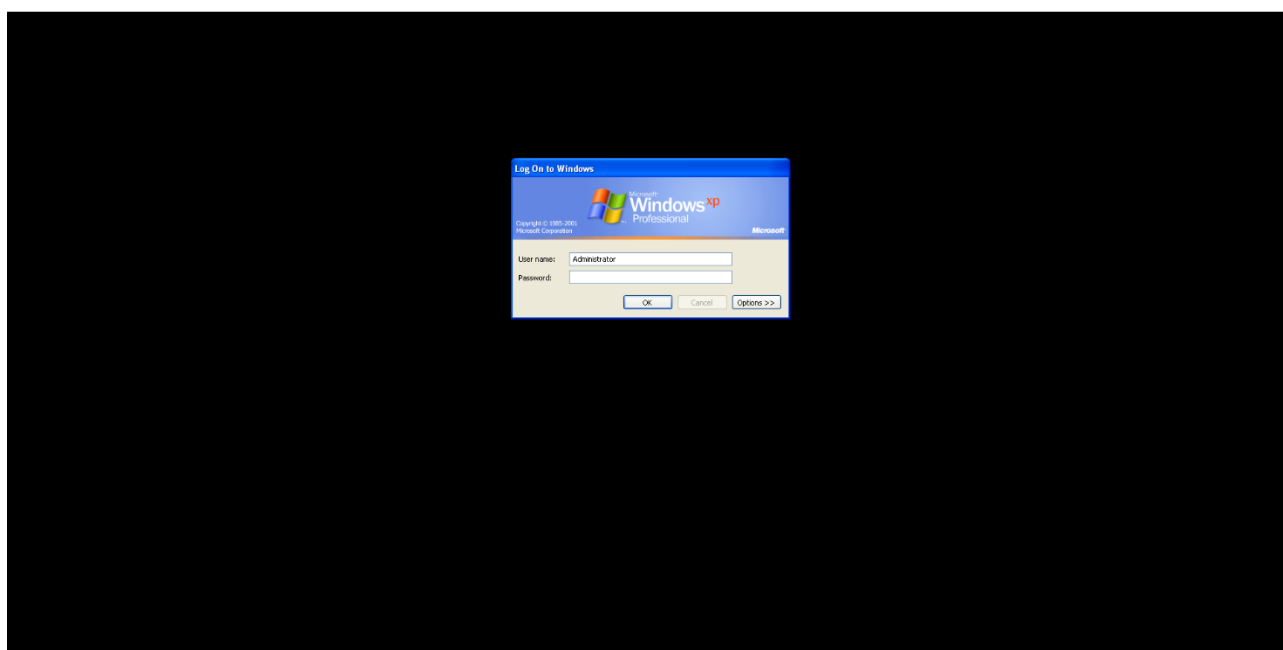
Analizzando invece il nostro caso, il comportamento della dll generata dal malware è il seguente.

Come anticipato in precedenza vediamo il comportamento del malware solo dopo il riavvio del sistema operativo, la prima anomalia la notiamo nella schermata di login, infatti essa sarà diversa.

LOGIN ORIGINALE

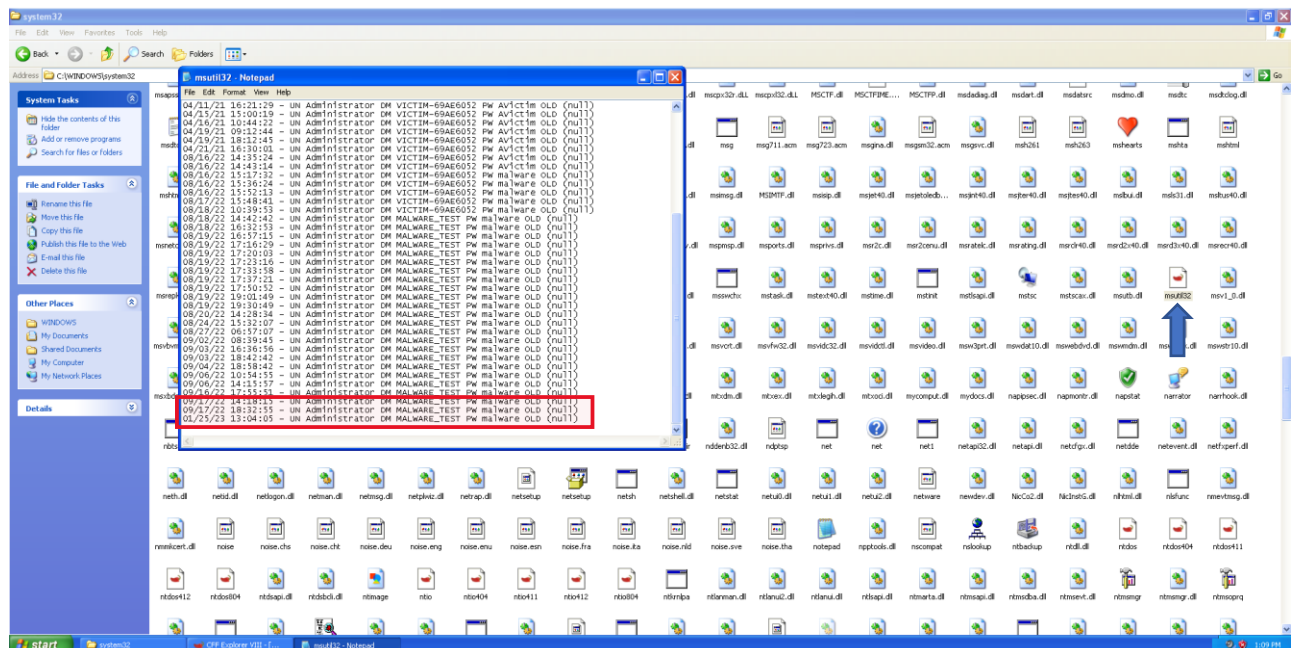
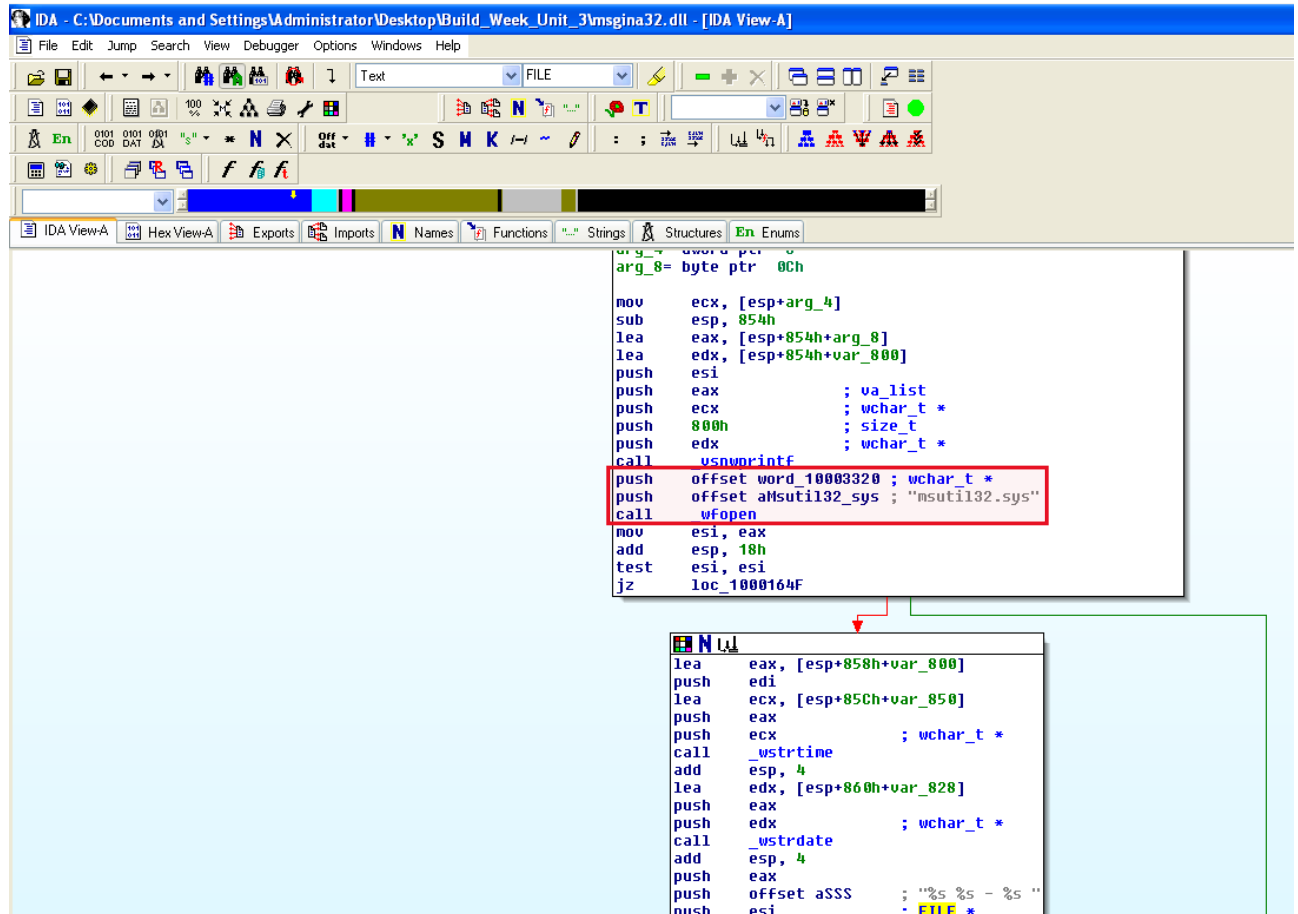


LOGIN MODIFICATO



Successivamente, dopo un po' di ricerche vediamo il comportamento del malware, appartenente appunto alla famiglia degli spyware.

Notiamo, infatti, che all'interno del path "%SystemRoot%\System32" viene scritto il file "msutil32.sys" dove vengono riportati i dati di login.



2. DIAGRAMMA DI FLUSSO FUNZIONALITA' MALWARE

