[<(‘.’<) (>’.’)>]Malware Analysis (͡° ͜ʖ ͡°)(͡° ͜ʖ ͡°)

Alessandro Morabito--Antonio Spirin--Daniele Morabito--Ernesto Robles--Stefano Castiglioni-- Giorgio Trovesi--Giuseppe Pariota

---

DAY1

Nel seguente report, effettueremo l’analisi di un malware, denominato **Malware_Build_Week_U3**, individuandone gli elementi costituenti e le possibili funzioni, giustificando quest’ultime con gli elementi evidenziati nell’analisi stessa.

Analisi strutturale del malware.

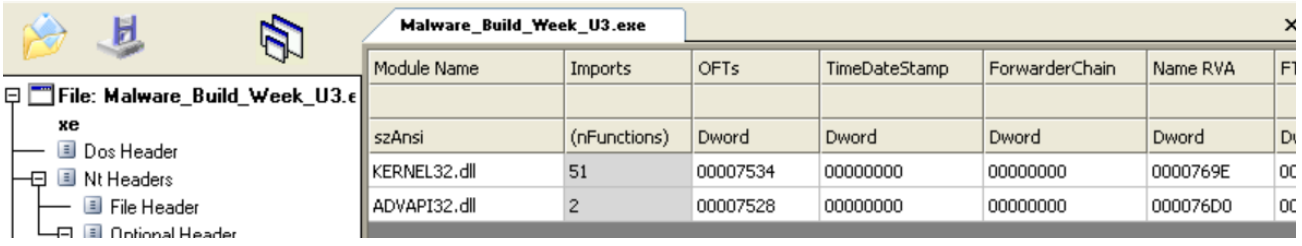
Innanzitutto, esaminiamo la struttura del malware per ricavarne le **sezioni** di cui è composto, come mostrate nell’immagine seguente.

property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	6BB361AB84E6EA32F545B128…	23FDE5162E5B17A6E440A46…	E433B4C400EFC11A593220E…	9D561586EEB5ECDA6C3214C…
entropy	6.225	3.770	0.601	4.154
file-ratio (92.31%)	46.15 %	7.69 %	23.08 %	15.38 %
raw-address	0x00001000	0x00007000	0x00008000	0x0000B000
raw-size (49152 bytes)	0x00006000 (24576 bytes)	0x00001000 (4096 bytes)	0x00003000 (12288 bytes)	0x00002000 (8192 bytes)
virtual-address	0x00401000	0x00407000	0x00408000	0x0040C000
virtual-size (47372 bytes)	0x00005646 (22086 bytes)	0x000009AE (2478 bytes)	0x00003EA8 (16040 bytes)	0x00001A70 (6768 bytes)
entry-point	0x00001487	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-
discardable	-	-	-	-
initialized-data	-	x	x	x
uninitialized-data	-	-	-	-
unreadable	-	-	-	-
self-modifying	-	-	-	-
virtualized	-	-	-	-
file	executable, offset: 0x00003249	n/a	n/a	executable, offset: 0x0000B070

Di seguito una rapida spiegazione delle sezioni scoperte:

- **.text**: contiene il codice eseguibile del programma.
- **.data**: contiene le variabili globali del programma, definite dal programmatore. Questa sezione può essere modificata dal programma stesso durante l'esecuzione.
- **.rdata**: sezione di sola lettura, contiene le costanti e le stringhe a cui fa riferimento il programma.
- **.rsrc**: contiene le risorse utilizzate dal programma, quali ad esempio icone e ulteriori dati.

Esaminiamo poi le **librerie** importate dal malware, fornendone una rapida overview. L'analisi delle librerie risulta fondamentale in fase di indagine in quanto può fornire importanti indizi riguardo il funzionamento del malware stesso, indicando in particolare le funzioni che questo utilizza e le parti del sistema che va a modificare.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FT
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	D
KERNEL32.dll	51	00007534	00000000	00000000	0000769E	00
ADVAPI32.dll	2	00007528	00000000	00000000	000076D0	00

- **kernel32.dll:** è una libreria che contiene le funzioni principali per l'interazione con il sistema operativo come la gestione della memoria o la manipolazione dei file.
- **advapi32.dll:** questa libreria contiene le risorse per interagire con i servizi ed i registri del sistema operativo.

Per quanto riguarda le **funzioni** richiamate dal software malevolo, di seguito le più interessanti al fine di capire il suo funzionamento.

Dword	Dword	Word	szAnsi
00007632	00007632	0295	SizeofResource
00007644	00007644	01D5	LockResource
00007654	00007654	01C7	LoadResource
00007664	00007664	00A3	FindResourceA

Al netto di librerie e funzioni prese in esame, possiamo immaginare che il malware si comporti come un **dropper**, caricando ed eventualmente eseguendo un ulteriore software contenuto al suo interno: sarà possibile confermare tale ipotesi solo dopo una più approfondita analisi statica e dinamica avanzata.

Parametri e variabili.

Attraverso l'analisi del codice assembly ricavato dal malware evidenziamo quali paramenti e variabili vengano rispettivamente passate e dichiarate all'interno della **funzione main()**, come mostrate nella prossima immagine.

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

hModule= dword ptr -11Ch
Data= byte ptr -118h
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h
```

Notiamo che sono presenti **quattro variabili** e **tre parametri**, rispettivamente con offset degli indirizzi di memoria **negativo** e **positivo** rispetto quello del registro **ebp**.

DAY2

Analisi del malware tramite codice assembly

Esamineremo di seguito una specifica funzione presente nel codice del malware, visibile nell'immagine seguente espressa in formato assembly.

```
text:0040101C                   push    80000002h       ; hKey
text:00401021                   call    ds:RegCreateKeyExA
text:00401027                   test    eax, eax
text:00401029                   jz      short loc_401032
text:0040102B                   mov     eax, 1
text:00401030                   jmp     short loc_40107B
```

La funzione interessata è **RegCreateKeyExA** all'indirizzo di memoria 00401021 ed richiamata nello stack dal **call**; la funzione ha lo scopo di creare (o aprire, se già presente) la chiave di registro specificata dal parametro corrispondente della funzione stessa. I parametri che accetta le vengono infatti passati attraverso i **push** alle righe superiori. In particolare, quella che le viene passata col push all'indirizzo di memoria 00401017 è la **subkey**, in questo caso il valore della chiave che viene passata alla funzione stessa.

Un altro interessante costrutto è presente agli indirizzi 00401027 e 00401029; abbiamo infatti un **test** tra due argomenti uguali, eax: essendo **test** equiparabile all'operazione booleana **AND**, il risultato di un'operazione avente gli operandi uguali sarà uguale all'operando stesso. Inoltre, è importante ricordare che l'operazione test modifica solo il **registro EFLAG**. Guardando alla riga successiva, il salto condizionale **jz** è effettuato solo se il test precedente imposta la Zero Flag (ZF = 1), ciò a sua volta avviene solo, in questo caso, se eax è uguale a 0.

```
text:00401027                   test    eax, eax
text:00401029                   jz      short loc_401032
```

Di seguito, a scopo esemplificativo, il codice assembly esaminato viene "tradotto" in **C**.

```
If (eax == 0) { jump to short_loc 00401032
```

Tornando all'analisi del codice assembly, la successiva funzione chiamata sullo stack di particolare interesse è **RegSetValueExA**, mostrata nella seguente immagine.

```
.text:0040103E                  push    offset ValueName ; "GinaDLL"
.text:00401043                  mov     eax, [ebp+hObject]
.text:00401046                  push    eax             ; hKey
.text:00401047                  call    ds:RegSetValueExA
```

Tra i vari parametri importati, possiamo notare **ValueName**, a cui è assegnato il valore **GinaDLL**; questa è la chiave di registro su cui lavorerà la funzione.

Considerando gli elementi analizzati finora, possiamo supporre che con questi ultimi passaggi il malware stia cercando di modificare le chiavi di registro per ottenere **persistenza** sul sistema o per "intaccare" il processo di **autenticazione** del sistema stesso.

DAY3

Analisi routine del malware

Procedendo con l'analisi, utilizziamo anche il software **OllyDBG**, di cui è mostrato un estratto nella prossima immagine, per esaminare il malware in un ambiente controllato in cui è possibile interrompere il flusso del codice a piacimento.

```
004010BD | . 50            | PUSH EAX                                  | ResourceType => "BINARY"
004010BE | . 8B0D 34804000 | MOV ECX,DWORD PTR DS:[408034]             | Malware_.00408038
004010C4 | . 51            | PUSH ECX                                  | ResourceName => "TGAD"
004010C5 | . 8B55 08       | MOV EDX,DWORD PTR SS:[EBP+8]              |
004010C8 | . 52            | PUSH EDX                                  | hModule
004010C9 | . FF15 28704000 | CALL DWORD PTR DS:[<&KERNEL32.FindResou   | FindResourceA
004010CF | . 8945 EC       | MOV DWORD PTR SS:[EBP-14],EAX             |
004010D2 | . 837D EC 00    | CMP DWORD PTR SS:[EBP-14],0               |
```

Tra gli elementi della routine presa in esame notiamo la funzione **FindResourceA** ed in particolare il parametro necessario **ResourceName**, a cui è assegnato il valore **TGAD**: questa è la porzione del malware a cui il codice accede per trovare un secondo codice malevolo da eseguire successivamente, probabilmente il **GinaDLL** visto in precedenza; considerando ciò e il continuo susseguirsi di chiamate di funzione **FindResource()**, **LoadResource()** e **SizeOfResource()** abbiamo una ulteriore conferma che l'eseguibile esaminato sia un **dropper**.

Tali informazioni, seppur non disponibili con analisi statica base in quanto questa non ci permette di indagare la sezione .**rsrc**, erano in parte visibili dal codice assembly di **IDA**, come mostrato nella seguente immagine, dove **lpName** prende come valore la stessa porzione **TGAD**.

```
text:004010C5                   mov     edx, [ebp+hModule]
text:004010C8                   push    edx             ; hMod; LPCSTR lpName
text:004010C9                   call    ds:FindResourceA        lpName          dd offset aTgad         ; DATA XREF: sub_401080+3E↑r
text:004010CF                   mov     [ebp+hResInfo], eax                                             ; "TGAD"
```

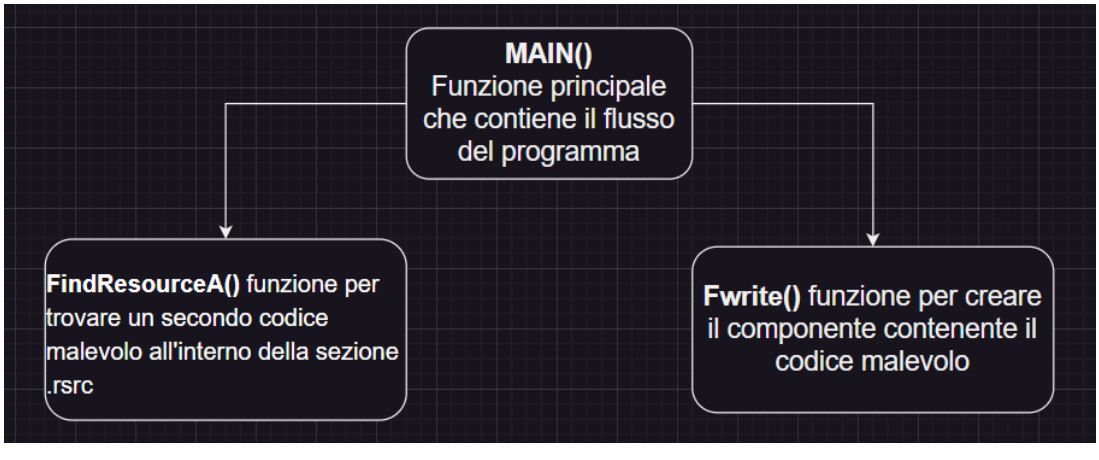
Eseguendo il malware viene creato con la funzione **Fwrite()** nella sua stessa cartella il file **msgina32.dll**, sul quale si può effettuare una analisi statica.

```
push    offset aMsgina32_dll_0 ; "msgina32.dll"
call    _fopen
```

```
push    eax             ; void *
call    _fwrite
```

Ancora una volta, possiamo ipotizzare che il malware interagisca con il processo di **autenticazione** dell'utente, ma non è possibile giungere ad alcuna conclusione certa sul suo scopo limitandoci ad analizzare le informazioni disponibili sulle funzioni esportate e librerie importate.

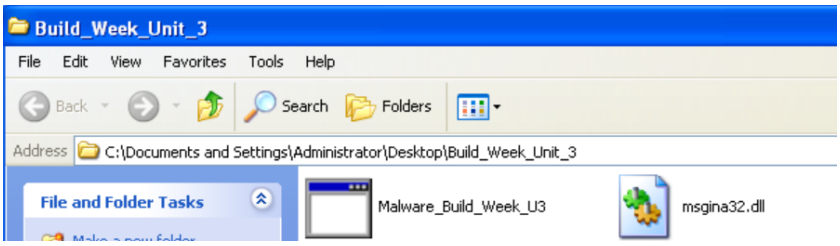
A scopo esemplificativo, di seguito è presente un diagramma di flusso che mostra i processi del malware fino al punto esaminato.



DAY4

Esame dei processi del malware

Come visto in precedenza, in seguito all'esecuzione del malware viene creato un file msgina32.dll, il quale è probabilmente una versione alterata della libreria presa di mira.



Per esaminare questo processo e, in generale, i processi generati dal malware, utilizziamo **Procmon**. Notiamo immediatamente il processo con cui viene creata la nuova chiave di registro, ossia **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\Currentversion\Winlogon**, e il successivo, evidenziato nella seguente immagine, in cui le viene assegnato il valore **GinaDLL**.

1940	RegOpenKey	HKLM	SUCCESS	Desired Access: Maximum Allowed
1940	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Diagnostics	NAME NO...	Desired Access: Read
1940	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ntdll.dll	NAME NO...	Desired Access: Read
1940	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\kernel32.dll	NAME NO...	Desired Access: Read
1940	RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	Desired Access: All Access
1940	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL	SUCCESS	Type: REG_SZ, Length: 520, Data:
1940	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon	SUCCESS	

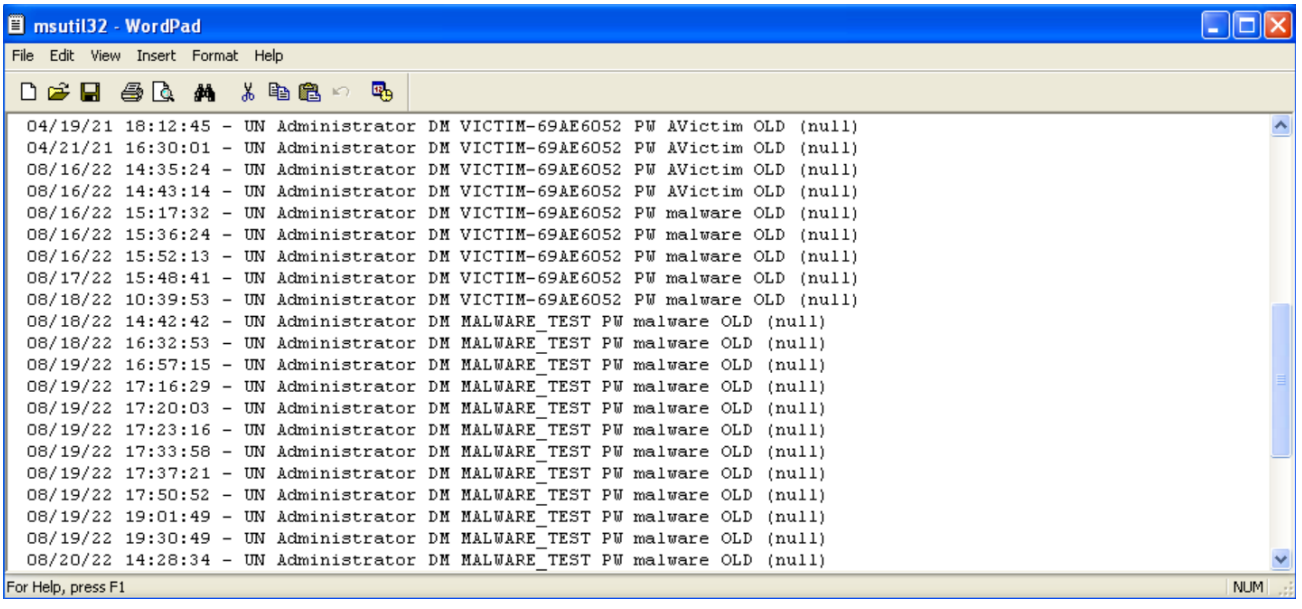
Vediamo infine la chiamata di sistema che ha creato il nuovo file nella cartella del malware.

Malware_Build_W...	1940	CreateFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	SUCCESS
Malware_Build_W...	1940	FileSystemControl	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	SUCCESS
Malware_Build_W...	1940	QueryOpen	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\Malware_Build_Week_U3.exe.Local	NAME NO...
Malware_Build_W...	1940	CreateFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS
Malware_Build_W...	1940	CreateFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	SUCCESS
Malware_Build_W...	1940	CloseFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3	SUCCESS
Malware_Build_W...	1940	WriteFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS
Malware_Build_W...	1940	WriteFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS
Malware_Build_W...	1940	CloseFile	C:\Documents and Settings\Administrator\Desktop\Build_Week_Unit_3\msgina32.dll	SUCCESS
Malware_Build_W...	1940	SetEndOfFileInfo...	C:\WINDOWS\system32\config\software.LOG	SUCCESS

DAY5

Conclusioni

Al netto delle azioni del malware, deduciamo che in fase di autenticazione il sistema userà la versione **alterata** della **libreria gina.dll**, andando probabilmente a **rubare le credenziali** inserite dall'utente. Il malware, inoltre, crea un **file testuale** chiamato **msutil.sys** sul quale registra gli input dell'utente stesso: nella seguente immagine, "Administrator" è il nome dell'utente connessosi mentre "malware" è la sua password.



Di seguito il frammento di codice assembly in cui si nota la creazione del documento di testo.

```
.text:1000158E        call    _vsnwprintf
.text:10001593        push    offset word_10003320 ; wchar_t *
.text:10001598        push    offset aMsutil32_sys ; "msutil32.sys"
.text:1000159D        call    _wfopen
.text:100015A2        mov     esi, eax
.text:100015A4        add     esp, 18h
.text:100015A7        test    esi, esi
```

In conclusione, mostriamo il **completo funzionamento del malware** in un diagramma di flusso di alto livello, rendendo evidenti tutti i passaggi che l'eseguibile effettua una volta avviato.