PROGETTO S10-L5

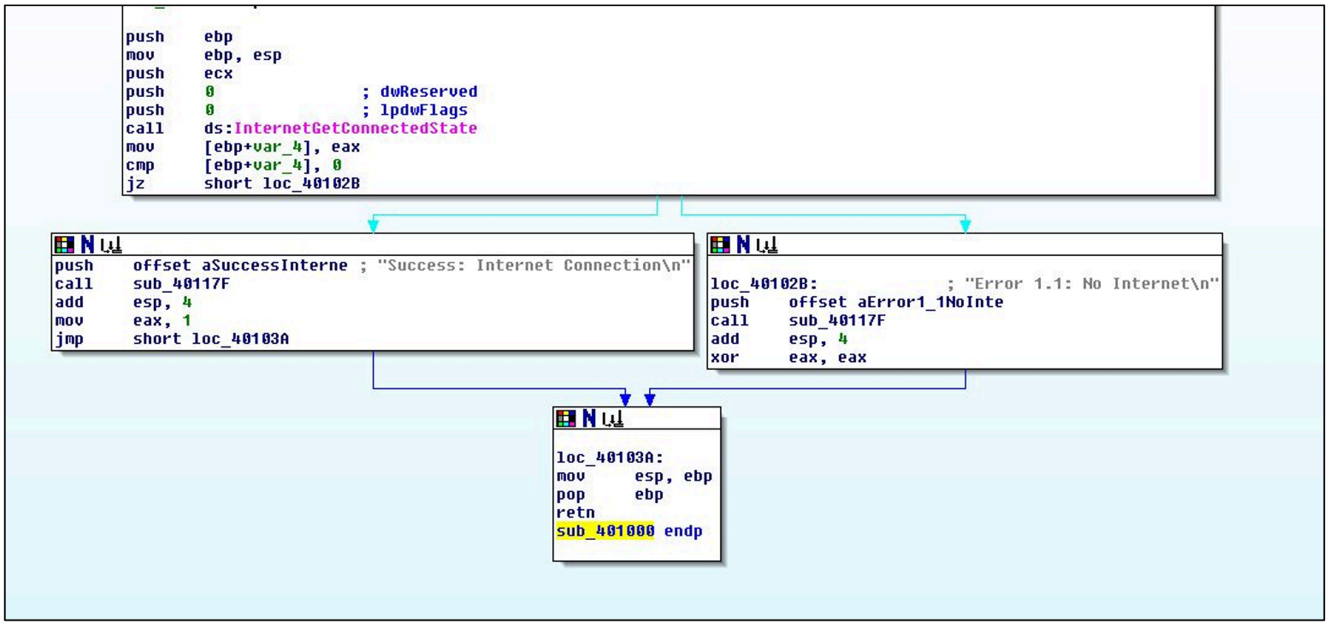
INDICE

1. Traccia

Con riferimento al file **Malware_U3_W2_L5** presente all'interno della cartella «**Esercizio_Pratico_U3_W2_L5**» sul desktop della macchina virtuale dedicata per l'analisi dei malware, rispondere ai seguenti quesiti:

- Quali librerie vengono importate dal file eseguibile?
- Quali sono le sezioni di cui si compone il file eseguibile del malware?

Figura 1



Con riferimento alla figura, risponde ai seguenti quesiti:

- Identificare i costrutti noti (creazione dello stack, eventuali cicli, costrutti)
- Ipotizzare il comportamento della funzionalità implementata

2. Riferimenti teorici

Analisi statica basica

I Malware (malicious software) includono una vasta gamma di programmi scritti per arrecare danno a sistemi informativi, spesso a scopo di lucro. L'analisi del malware o «**malware analysis**» è l'insieme di competenze e tecniche che permettono ad un analista della sicurezza informatica di indagare accuratamente un malware per studiare e capire esattamente il suo comportamento al fine di rimuoverlo dal sistema. Queste competenze sono fondamentali per i membri tecnici del CSIRT durante la risposta agli incidenti di sicurezza.

Durante lo studio dell'analisi dei malware, incontreremo due tecniche principali di analisi:

- L'analisi statica
- L'analisi dinamica

Nel corso del nostro studio ci troveremo dunque ad affrontare:
Analisi statica basica: l'analisi statica basica consiste nell'**esaminare un eseguibile senza vedere le istruzioni che lo compongono**. Lo scopo dell'analisi basica statica è di confermare se un dato file è malevolo e fornire informazioni generiche circa le sue funzionalità. L'analisi statica basica è sicuramente la più intuitiva e semplice da mettere in pratica, ma risulta anche essere la più inefficiente soprattutto contro malware sofisticati.
Analisi dinamica basica: l'analisi dinamica basica presuppone l'esecuzione del malware in modo tale da osservare il suo comportamento sul sistema infetto al fine di rimuovere l'infezione. I malware devono essere eseguiti in ambiente sicuro e controllato in modo tale da eliminare ogni rischio di arrecare danno a sistemi o all'intera rete. Così come per l'analisi statica basica, l'analisi dinamica basica è piuttosto semplice da mettere in pratica, ma non è molto efficace quando ci si trova ad analizzare malware sofisticati.
Analisi statica avanzata: l'analisi statica avanzata presuppone la conoscenza dei fondamenti di «reverse-engineering» al fine di identificare il comportamento di un malware a partire dall'analisi delle istruzioni che lo compongono. In questa fase vengono utilizzati dei tool chiamati «disassambler» che ricevono in input un file eseguibile e restituiscono in output il linguaggio «assembly». Vedremo i concetti di reverse-engineering e il linguaggio assembly prima di affrontare lo studio dell'analisi statica avanzata.
Analisi dinamica avanzata: l'analisi dinamica avanzata presuppone la conoscenza dei debugger per esaminare lo stato di un programma durante l'esecuzione. I debugger saranno introdotti prima dello studio dell'analisi dinamica avanzata.

Lo studio e la comprensione del comportamento esatto di un malware è un compito piuttosto complicato. Tuttavia, può essere semplificato identificando il tipo di malware che si sta analizzando, ad esempio: se siete di fronte ad un malware che si mette in ascolto su una porta TCP e garantisce una shell a chi si connette, possiamo pensare che si tratti di una **backdoor**.
Allo stesso modo, un malware che contatta un dominio per scaricare un altro file eseguibile, potrebbe essere un «**downloader**», ovvero un malware che scarica altri malware.

Capire preventivamente il tipo di malware da alcuni caratteri generali può dunque aiutare nella comprensione del comportamento generale.

Analisi statica basica

Quando si analizza un potenziale malware il prima passo da fare è assicurarsi che sia di fatto un malware. Nello studio della Unit 2 abbiamo detto che ogni file ha una propria firma (file signature), di conseguenza anche i malware hanno una propria firma.
Per capire se stiamo analizzando un malware, potremmo **controllare nei database degli antivirus se la firma del nostro malware è nota.**
Siti come **VirusTotal** (https://www.virustotal.com/gui/home/upload), ci permettono di caricare un file eseguibile e controllare la sua reputazione in base ad un numero variabile ma consistente di software antivirus.
Per utilizzare VirusTotal, una volta connessi al link cliccare su «**choose file**» e caricare il file che si vuole controllare.
VirusTotal calcolerà per prima cosa la firma del software e poi controllerà se la stessa è già presente nei database dei software antivirus, e la sua eventuale categorizzazione come «malware».

Alternativamente si può calcolare «**l'hash» di un malware, ovvero una stringa alfanumerica unica per identificare un file** (abbiamo già incontrato gli hash quando abbiamo parlato del password cracking, dove in quel caso abbiamo visto gli hash delle password).
Per calcolare l'hash di un file, possiamo utilizzare l'utility «md5deep» che nella macchina virtuale è nella **cartella «md5deep-4.3»** presente di default sul vostro desktop.

Per l'utilizzo dei tool da riga di comando, dovremmo seguire gli step così di seguito:

- Aprire il «command prompt» (presente sul desktop)
- Spostarsi con il comando «cd» nella cartella contenente l'eseguibile da utilizzare
- Eseguire il comando seguito dai suoi parametri

Finora abbiamo parlato dei file eseguibili, ma non li abbiamo ancora definiti. **Windows utilizza per la maggior parte dei file eseguibili il formato PE, Portable Executable**.
Il formato PE al suo interno contiene delle informazioni necessarie al sistema operativo per capire come gestire il codice del file, come ad esempio le librerie.
Abbiamo rapidamente discusso delle librerie (anche chiamate moduli) nella Unit 1, definendolo come insieme di funzioni. Quando un programma ha bisogno di una funzione «chiama» una libreria al cui interno è definita la funzione necessaria.
Si dice anche che il programma ha importato una libreria (ricordate la sintassi iniziale dei programmi Python: import «modulo»).

La tabella di seguito riporta alcune delle librerie importate dinamicamente più utilizzate e la loro descrizione.

Kernel32.dll: libreria piuttosto comune che contiene le funzioni principali **per interagire con il sistema operativo, ad esempio: manipolazione dei file, la gestione della memoria.**
Advapi32.dll: libreria che contiene le funzioni **per interagire con i servizi ed i registri del sistema operativo Microsoft.**
WSock32.dll e Ws2_32.dll: sono librerie che **contengono le funzioni di network, come le socket, le funzioni connect, bind.** Ogni **malware che utilizza funzionalità di rete** caricherà certamente una di queste librerie.
Wininet.dll: libreria che contiene le funzioni per l'**implementazione di alcuni protocolli di rete come HTTP, FTP, NTP.**
Gdi32.dll: libreria che contiene le funzioni per l'**implementazione e manipolazione della grafica e dei suoi effetti.**
MSVCRT.dll: libreria che contiene funzioni per la **manipolazione di stringhe, allocazione di memoria e altro come chiamate per input/output in stile linguaggio C.**

Come avete probabilmente notato, la figura vista in precedenza (CFF Explorer/section headers) riporta non solo il nome delle sezioni ma anche altre importanti informazioni, come ad esempio:
Virtual size: indica lo spazio allocato per la sezione **durante il processo di caricamento** dell'eseguibile in memoria.
Raw size: indica lo spazio occupato dalla sezione **quando è sul disco**.

Utilizzando un set di tool piuttosto semplici, possiamo recuperare importanti informazioni circa un eseguibile, ed iniziare ad avere un'idea sul suo comportamento.
Come abbiamo detto inizialmente, tuttavia, l'analisi statica è un processo piuttosto semplice ma altresì inefficace contro i tipi di malware più avanzati che tendono per esempio ad oscurare il nome delle sezioni, oppure a configurare le sezioni con un nome senza un senso logico.
Alcuni malware utilizzano ad esempio il caricamento delle librerie durante l'esecuzione (runtime import) nascondendo di fatto all'analisi statica le funzioni e le librerie importate. Questi malware sono riconoscibili in quanto hanno generalmente poche entry nella sezione import, e tra esse figurano le funzioni «LoadLibrary e GetProcAddress» che vengono appunto utilizzate per caricare funzioni addizionali durante l'esecuzione.

Analisi dinamica basica

L'analisi dinamica comprende tutte quelle attività di analisi che presuppongono l'**esecuzione del malware in un ambiente dedicato.**
In termini di processo l'analisi dinamica basica è generalmente effettuata dopo l'analisi statica basica, per sopperire ai limiti dell'analisi statica ed avere una maggiore visibilità sulle attività e il comportamento del malware in esame.
Al contrario dell'analisi statica, l'analisi dinamica permette di **osservare e studiare le vere funzionalità di un malware in esecuzione** su un sistema.

Per avviare un malware basta avviare il suo eseguibile con un doppio click. Per quanto ci siano diversi tipi di file eseguibili in formato PE, all'interno di questo corso ci concentreremo su file con estensione «**.exe**». L'estensione «.exe» rappresenta lo standard per quanto riguarda gli eseguibili su sistemi Windows.
Di conseguenza, per avviare un malware basterà eseguire un doppio click sul relativo file eseguibile con formato «.exe».

Una volta avviato un malware, è importante capire quali siano le sue **attività sul sistema**. Un malware potrebbe infatti:

- **Aprire un determinato file per cifrarlo**
- **Iniziare una comunicazione verso internet**
- **Modificare una chiave di registro Windows**
- **Modificare le impostazioni di sistema**
- **Lanciare servizi e processi aggiuntivi**

E molto altro.

Per monitorare i comportamenti dei malware in esecuzione, ci vengono in supporto diversi tool.

Tornando all'interfaccia del tool, possiamo notare che il pannello principale è costituito da un insieme configurabile di colonne, contenenti generalmente:

- **Time:** Tempo di cattura
- **Process Name:** il nome del processo
- **PID:** process ID, identificativo univoco dei processi su sistemi Windows
- **Operation:** ovvero l'azione effettuata. Ad esempio la riga 1 della figura riporta che il processo lsass.exe sta provando ad aprire (RegOpenKey) una chiave di registro. L'operazione è a tutti gli effetti una chiamata ad una funzione contenuta in una data libreria. Nelle fasi iniziali dello studio, la maggior parte delle funzioni saranno sconosciute. Imparerete a conoscerle con l'esercitazione pratica. Qualora doveste incontrare una funzione non nota, potete cercarla su internet. Il consiglio è di prendere nota di tutte le funzioni ed il loro utilizzo.
- **Path:** indica il percorso dove si sta concretizzando l'azione. Riprendendo l'esempio della prima riga, il path ci indica che la chiave di registro che si sta aprendo è al path «HKLM\SECURITY\Policy»
- **Result:** il risultato dell'azione (dell'operazione)
- **Detail:** dettaglio della richiesta (dell'operazione)

Tra le categorie più significative troviamo:
HKEY_CURRENT_USER: include le impostazioni e preferenze di sistema dell'utente che è attualmente connesso alla macchina
HKEY_LOCAL_MACHINE: include le impostazioni comuni per tutti gli utenti del sistema indipendentemente dalle loro preferenze
HKEY_USERS: raggruppa le impostazioni di tutti gli utenti connessi al sistema

I nomi delle categorie delle chiavi di registro sono molto spesso abbreviate, per cui HKEY_CURRENT_USER diventa **HCU**, HKEY_LOCAL_MACHINE diventa **HKLM** e HKEY_USERS diventa **HKU**.

La maggior parte dei malware in esecuzione genera attività di rete, come ad esempio contattare un dominio per scaricare ulteriori malware, oppure caricare un file su un server remoto o ancora contattare un «command and control» server per ricevere comandi. Di conseguenza **monitorare i flussi di rete durante l'esecuzione** di un malware è fondamentale.
Uno dei tool più utili per il monitoraggio delle rete è **Wireshark**, che per Windows è installato di default sulla macchina dedicata all'analisi dei malware, ed al netto di piccole modifiche grafiche, valgono gli stessi concetti visti per la versione su Kali Linux.

Assembly x86

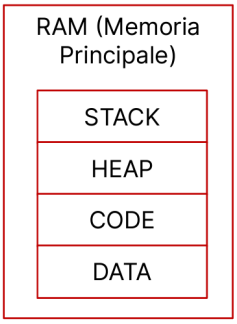
L'analisi statica avanzata presuppone la conoscenza di un particolare tipo di linguaggio, chiamato **linguaggio Assembly**. Il linguaggio Assembly **è univoco per una data architettura di un PC**, ma cambia da architettura ad architettura come vedremo in seguito.
L'analista di sicurezza durante l'**analisi statica** utilizzerà dei tool chiamati «**Disassembler**» **che sono programmati per tradurre le istruzioni binarie eseguite dalla CPU in istruzioni** più leggibili dall'uomo, che è proprio il linguaggio Assembly.
Dunque la conoscenza del linguaggio **Assembly servirà per «leggere» le istruzioni eseguite dalla CPU in formato leggibile dall'uomo.**

Per quanto riguarda i linguaggi, possiamo identificare 6 livelli di astrazione:
Hardware: qui risiede il linguaggio macchina, e le istruzioni / operazioni sono effettuate per mezzo di livelli logici AND, OR, XOR basati su circuiti elettrici.
Microcodice: traduce le istruzioni dei livelli superiori per l'hardware specifico.
Codice Macchina: consiste di istruzioni scritte in «opcodes», ovvero caratteri esadecimali che istruiscono il processore su cosa fare.
Linguaggi di basso livello: includono istruzioni leggibili all'uomo specifiche per una data architettura di un calcolatore. Il linguaggio di basso livello più comune è l'**assembly** che viene ampiamente utilizzato nell'analisi dei malware.
Linguaggi di alto livello: includono istruzioni molto comprensibili dall'uomo e rendono semplice le logiche di programmazione. I linguaggi come il **C ed il C++** sono linguaggi di alto livello che forniscono un'astrazione del linguaggio assembly sottostante.
Linguaggi interpretati: sono i linguaggi di livello più alto, il codice di questi linguaggi non viene compilato in linguaggio macchina, ma piuttosto è tradotto in «bytecode» specifico di quel linguaggio che viene poi eseguito da un programma detto «interprete». Il **Python** è un linguaggio interpretato.

Architettura e Memoria

Abbiamo detto in precedenza che l'assembly è un linguaggio che dipende dall'architettura del calcolatore. Infatti esso si riferisce ad una famiglia di linguaggi di programmazione che differiscono tra loro in base all'architettura del processore, che ne definisce il set di istruzioni.
Tra le architetture più note troviamo x86,64, ARM, MIPS, PowerPC.
Quello che ci interessa sapere è che **i processori possono essere a 32 o 64 bit.**
Che vuol dire che un processore è 32 o 64 bit?
Senza scendere troppo in tecnicismi, il numero di bit indica l'ammontare di informazioni che la **CPU** riesce a gestire per ogni operazione.

La memoria principale per un singolo programma può essere divisa in diverse sezioni:
Data: contiene generalmente variabili del programma che non cambiano durante l'esecuzione come variabili statiche oppure variabili globali
Code: questa sezione contiene le istruzioni da eseguire
Heap: viene utilizzato per l'allocazione di memoria dinamicamente durante l'esecuzione di un programma
Stack: viene utilizzato per le variabili locali ed i parametri delle funzioni e per supportare il flusso del programma



Introduzione al linguaggio Assembly

Dopo aver rapidamente rivisto alcuni concetti sulle architetture ed la memoria possiamo passare ad introdurre i concetti del linguaggio Assembly per x86.
La base del linguaggio Assembly sono le **istruzioni**. Nel linguaggio Assembly le istruzioni sono costituite da due parti:

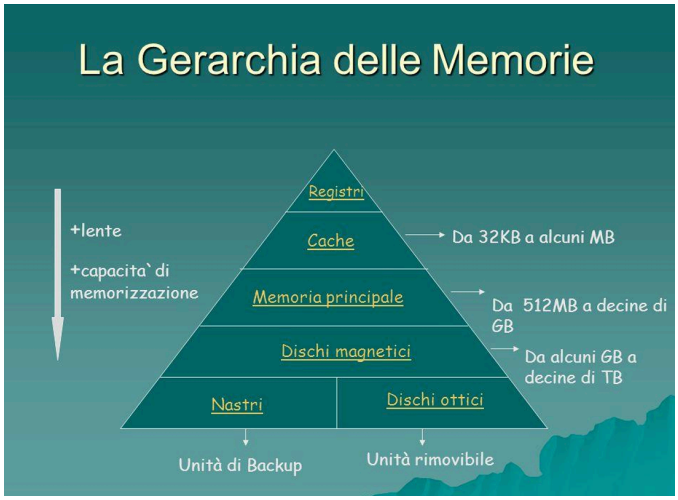
- **Un codice mnemonico**, ovvero una parola che identifica l'istruzione da eseguire
- **Uno o più operandi** (per alcuni codici mnemonici, non è necessario l'operando come vedremo a breve), che identificano le variabili o la memoria oggetto dell'istruzione.

Si possono utilizzare tre tipi diversi come **operando**:

- **Un valore**, come ad esempio il numero 5. Attenzione: generalmente i valori immediati non sono scritti in formato decimale ma bensì in formato esadecimali nella forma 0xYY dove YY rappresenta la versione **esadecimale** del numero decimale. Per semplicità di comprensione continueremo ad utilizzare nella teoria la forma decimale dei numeri, ma ricordate che in pratica, il numero che vedrete nella **forma 0xYY** è un numero scritto in forma esadecimale. Ricordate di convertire in decimale di conseguenza.
- **Uno dei registri** messi a disposizione dalla CPU che vedremo a breve
- **Un indirizzo di memoria** che contiene un valore di interesse

Si può riassumere il concetto visto sopra nella piramide nella figura che prende il nome di «**Gerarchia delle memorie**». La regola generale è: più rapido l'accesso alla memoria, minore sarà la capacità di memorizzazione.

Ad esempio, i **registri** che sono il tipo di memoria ad accesso più rapido possono contenere informazioni nell'ordine dei **bit**, mentre i **dischi rigidi** come gli hard disk o gli SSD possono contenere **terabyte** di informazioni.
A memorie ad accesso più rapido è anche associato un costo maggiore.



ISTRUZIONE	DESCRIZIONE
sub eax, 5	Sottrae il valore 5 dal registro EAX
add eax, ebx	Somma il valore contenuto in EBX ad EAX e salva il risultato in EAX
inc ecx	Incrementa il registro ecx di 1, è uguale a add ecx, 1
dec edx	Decrementa il registro edx di 1, è uguale a sub edx, 1

ISTRUZIONE	DESCRIZIONE
mul 5	Moltiplica il valore contenuto in EAX per 5 e salva il risultato nel registro EAX.
div 5	Divide il contenuto del registro EAX per 5 e salva il risultato nel registro EAX ed il resto della divisione nel registro EDX.

ISTRUZIONE	CASO DI UTILIZZO FREQUENTE	DESCRIZIONE
xor	xor eax, eax	Inizializza a 0 il registro EAX. Infatti l'operatore logico tra due bit identici restituisce sempre 0. Restituisce 1 solamente nel caso in cui i bit su cui opera sono diversi
or	or eax, 5	Restituisce l'OR logico tra i bit del registro EAX e la forma binaria del numero 5. Aggiorna poi EAX con il risultato dell'operazione
and	and eax, 5	Restituisce l'AND logico tra i bit di EAX e la forma binaria del numero 5. Aggiorna poi EAX con il risultato dell'operazione

CMP	ZF	CF
Destinazione = sorgente	1	0
Destinazione < sorgente	0	1
Destinazione > sorgente	0	0

ISTRUZIONE	DESCRIZIONE
jz loc	Salta alla locazione di memoria specificata se ZF = 1
jnz loc	Salta alla locazione di memoria specificata se ZF non è settato ad 1, ovvero è 0
je loc	Simile a jz, ma viene comunemente utilizzato dopo «cmp». Salta alla locazione di memoria specificata se gli operandi di «cmp» sono uguali
jne loc	Simile a jnz, utilizzato comunemente dopo «cmp». Salta alla locazione specificata se gli operandi differiscono tra di loro
jg loc	Salta alla locazione specificata se la destinazione è maggiore della sorgente nell'istruzione «cmp»
jge loc	Salta alla locazione specificata se la destinazione è maggiore o uguale della sorgente nell'istruzione «cmp»

Costrutti C - Assembly X86

Variabili Globali e locali

Come sappiamo dallo studio del linguaggio C, **le variabili globali sono variabili che non sono definite nel contesto di una funzione** precisa ma bensì vengono dichiarate globalmente e sono di conseguenza accessibili da tutte le funzioni di un programma. D'altro canto, **le variabili locali sono definite all'interno di una funzione** e pertanto sono accessibili solamente all'interno di quella specifica funzione. Se si tenta infatti di accedere o leggere una variabile localmente definita al di fuori del suo contesto di definizione, il compilatore ci restituirà un errore.

In Assembly, una variabile globale è trattata in maniera totalmente differente da una variabile locale. Infatti **mentre una variabile locale viene definita all'interno dello stack** di una funzione e per accedervi si utilizzano come referenze i puntatori dello stack (EBP in genere), **una variabile globale verrà letta utilizzando un indirizzo di memoria.** Infatti, se ricordate le sezioni della memoria che abbiamo visto nelle lezioni scorse, abbiamo detto che la sezione «**data**» **viene utilizzata per le variabili globali**, mentre le variabili locali sono salvate sullo stack.

Operazioni aritmetiche

Abbiamo visto nella lezione precedente le istruzioni per la gestione delle operazioni aritmetiche in Assembly.
La figura mostra delle semplici funzioni aritmetiche in Assembly, provate a «tradurle» in codice C.
Le prime due istruzioni sono delle «mov», gli argomenti delle due «mov» sono una variabile locale, in quanto referenziata sullo stack da EBP, ed un numero in formato decimale.

Ciclo if

I programmatori utilizzano i cicli if per modificare l'esecuzione del programma sulla base di determinate condizioni. I cicli IF sono piuttosto diffusi in C ed è quindi opportuno sapere come riconoscere il loro «equivalente» in Assembly.
La figura riporta la sintassi di un generico ciclo if in C, dove la scelta è presa in base alla condizione di uguaglianza tra le due variabili «x» ed «y».

Statement SWITCH
Il costrutto «switch» che abbiamo visto nelle lezioni introduttive del linguaggio C viene utilizzato per prendere decisioni in base al valore di una determinata variabile.
A titolo di esempio, la famiglia di malware che ricadono nella categoria delle backdoor, utilizza costrutti di tipo «switch» per consentire di eseguire localmente alla macchina una serie di azioni o comandi basati sul valore di una variabile. La figura di seguito mostra uno pseudocodice di una backdoor che mostra all'utente connesso una serie di comandi da poter eseguire sulla base del valore «command».

Il codice di un costrutto switch in C è mostrato nella figura a lato. Un blocco di istruzioni viene eseguito in base al valore della variabile «i». I costrutti C sono compilati (dal compilatore) in base a delle logiche di miglioramento prestazioni generalmente in due modi principali:

- Utilizzando una sintassi simile ad un **ciclo IF** (IF-style)
- Utilizzando le «**jump table**»

Ciclo for
I loop sono un altro costrutto molto usato dai programmatori per ripetere delle azioni / istruzioni un numero definito di volte.

- I loop, come il ciclo for in C, hanno quattro componenti principali:
- Il valore di inizio, anche detto inizializzazione
- Il valore di paragone, anche detto comparison
- Le istruzioni da eseguire, il corpo del loop
- L'incremento/decremento della variabile utilizzata per il ciclo

Ciclo While
Proseguiamo l'analisi dei costrutti C in Assembly con il prossimo ciclo: il while.
Il ciclo while viene frequentemente utilizzato all'interno del codice dei malware per creare dei loop. La particolarità del ciclo while è che il blocco di istruzioni viene eseguito ripetutamente e fintanto che una data condizione risulti vera.
I cicli while in assembly sono piuttosto simili ai cicli for. La figura mostra un generico ciclo while scritto in C, dove il blocco di codice viene eseguito fino a che «i» è minore di 10.

Chiamate di funzione
Durante le scorse lezioni abbiamo visto come una funzione può «chiamare» una seconda funzione per svolgere un determinato task.
Mentre la sintassi per le chiamate di funzioni in C è piuttosto immediata, è importante identificare e capire come esse avvengono nel linguaggio Assembly per la CPU x86.
Ci sono diversi tipi di convenzioni per le chiamate di funzione:

- Stdcall
- Fastcall
- Cdecl

Ma cosa significa esattamente creare e rimuovere lo stack dalla memoria?
Ricordiamo che lo stack è una porzione di memoria dedicata per il salvataggio delle variabili locali di una data funzione. Esso viene definito dai puntatori allo stack EBP che punta alla sua base ed ESP che punta alla cima. Di seguito le istruzioni in Assembly per creare lo stack di una funzione, e sono piuttosto standard al netto del valore sottratto ad ESP nella terza istruzione che rappresenta lo spazio destinato alle variabili locali.

00401020 push ebp
00401021 mov ebp, esp
00401023 sub esp, oCh

Allo stesso modo, le tre istruzioni seguenti sono utilizzate per rimuovere i piatti dal carrello, e quindi eliminare lo stack una volta che una determinata funzione ha finito il suo compito.

00401053 mov esp, ebp
00401055 pop ebp

3. Procedimento

ANALISI STATICA BASICA

In riferimento al file eseguibile “Malware_U3_W2_L5” presente nella macchina virtuale dedicata all’analisi dei malware ho eseguito un’analisi statica basica, rispondendo ai quesiti.

QUALI LIBRERIE VENGONO IMPORTATE DAL FILE ESEGUIBILE?

In un’analisi statica basica si cerca di determinare se il file è malevolo o meno senza eseguirlo. Ci sono diversi metodi per analizzare un virus, ma in questo specifico caso, per vedere quali librerie vengono importate utilizzo il programma “CFF Explorer”. Una volta caricato il file clicco sulla sezione “import directory” e vedrò le relative librerie.

Librerie Importate dal file eseguibile
Possiamo controllare le librerie importate del file eseguibile utilizzando CFF Explorer, scegliendo «**Import directory**» dal pannello principale. Le librerie importate dal file eseguibile sono:
Kernel32.dll
WININET.dll

Malware_U3_W2_L5.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000065EC	N/A	000064DC	000064E0	000064E4	000064E8	000064EC
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	44	00006518	00000000	00000000	000065EC	00006000
WININET.dll	5	000065CC	00000000	00000000	00006664	000060B4

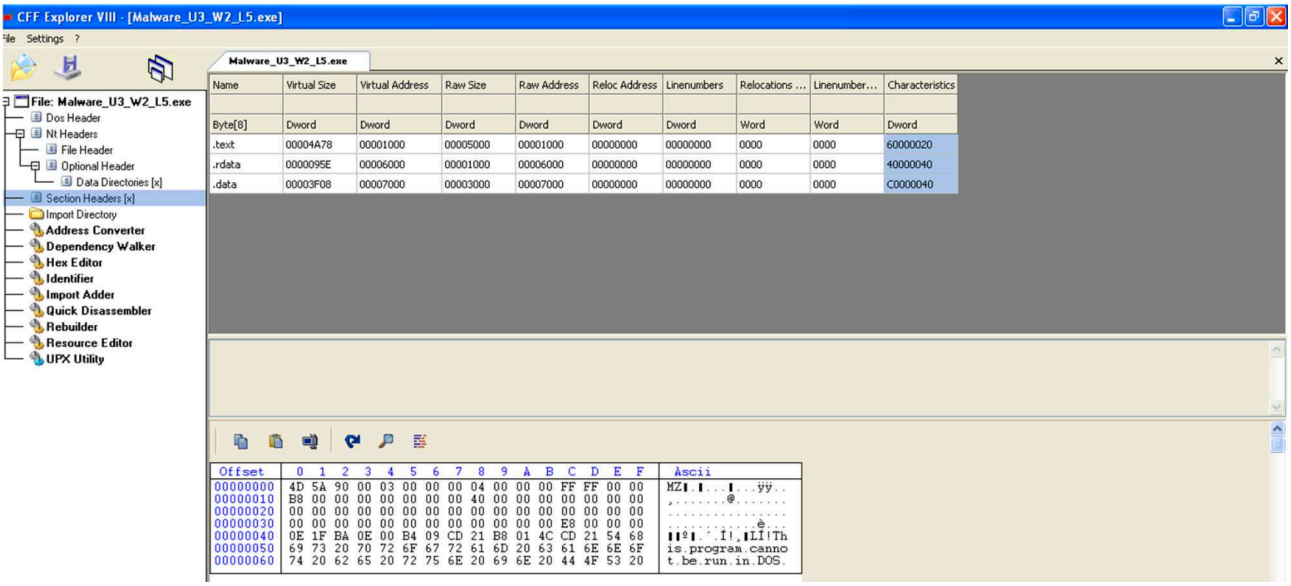
In questo caso le librerie sono due:

Kernel32.dll: si utilizza per interagire con il sistema operativo ad esempio: manipolazione dei file, gestione della memoria e così via.

Wininet.dll: contiene funzioni per implementare alcuni protocolli di rete, ad esempio HTTP, FTP, NTP.

QUALI SONO LE SEZIONI DI CUI SI COMPONE IL FILE ESEGUIBILE DEL MALWARE?

Le sezioni di cui si compone il file eseguibile si possono vedere sempre sul programma "CFF Explorer".
In questo caso, le sezioni sono le seguenti:

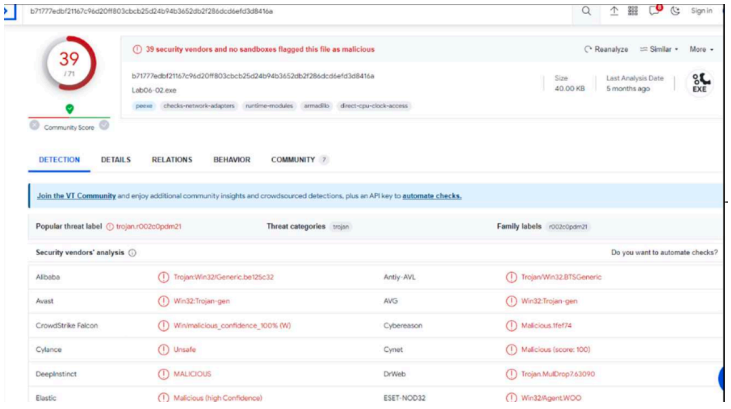


.text: contiene le righe di codice che la CPU eseguirà una volta che il software verrà avviato. Di solito è l'unica sezione di un file eseguibile dalla CPU.

.rdata: include informazioni sulle librerie e funzioni importate ed esportate dall'eseguibile.

.data: contiene dati e variabili globali del programma eseguibile.

Per avere maggiore conferma che effettivamente si tratti di un malware si può estrapolare l'hash del file e caricarlo su "Virus Total" (tool che ci permette di controllare la reputazione del file in questione). L'hash si può vedere sempre tramite "CFF Explorer", oppure calcolarlo con "md5deep".

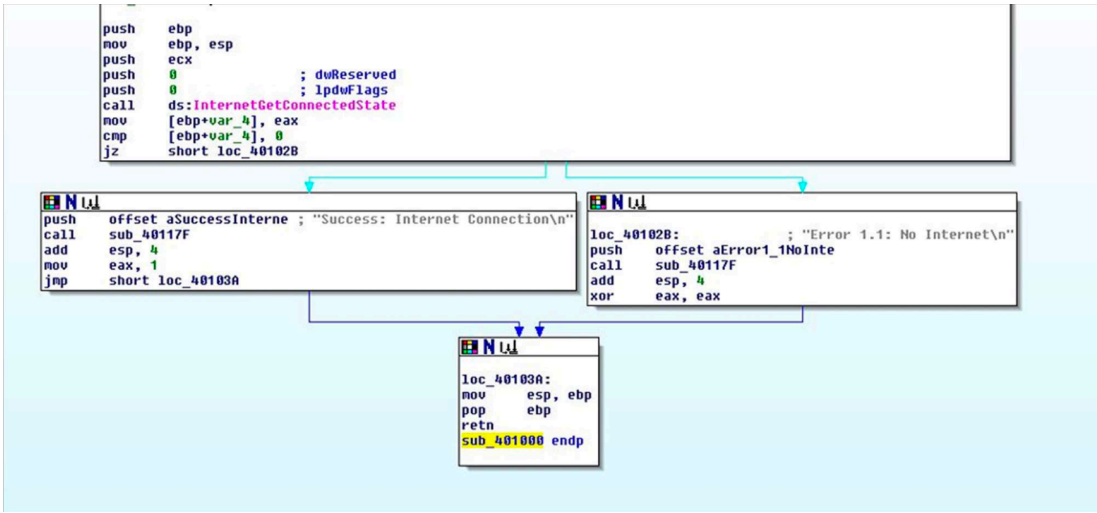


Il tool ha confermato che si tratti di un malware, in particolare di un trojan, un malware che si nasconde in file innocui e che si attiva quando la vittima apre il file.

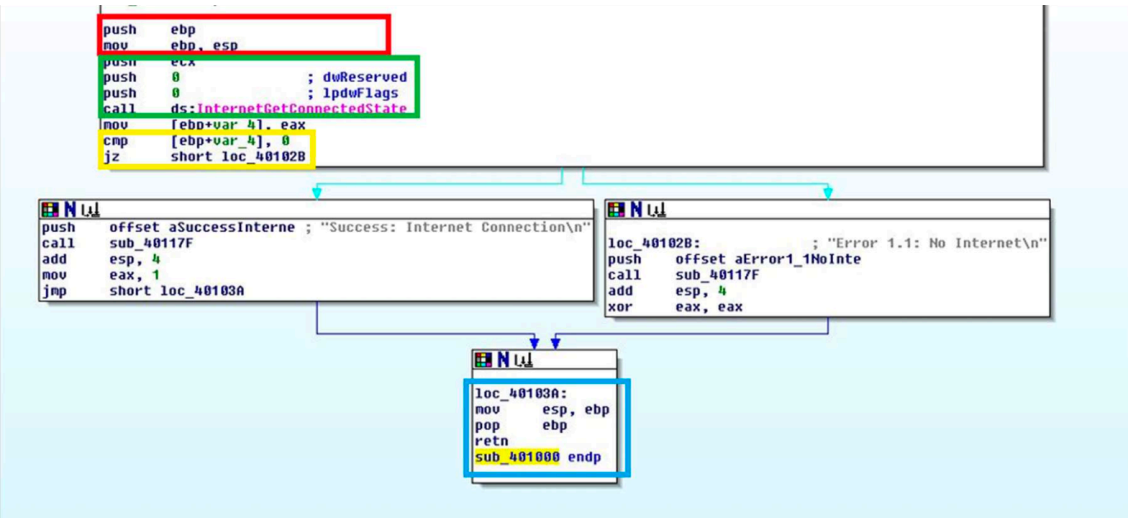
CODICE ASSEMBLY

L'assembly è un linguaggio utile per l'analisi statica avanzata. I programmi che traducono il linguaggio macchina in assembly si chiamano disassembler.

In questo caso il compito da svolgere è quello di identificare i costrutti noti e ipotizzare il comportamento della funzionalità implementata con riferimento alla seguente figura:



1. IDENTIFICARE I COSTRUTTI NOTI



Il riquadro rosso prevede la creazione dello stack
Il riquadro verde prevede la chiamata di funzione
Il riquadro giallo prevede il costrutto IF
Il riquadro azzurro prevede la chiusura dello stack

2. IPOTIZZARE IL COMPORTAMENTO DELLA FUNZIONALITA' IMPLEMENTATA

Questo estratto di codice controlla lo stato della connessione Internet. Quindi si verifica una condizione simile a un'istruzione IF in linguaggi di programmazione ad alto livello. Questo lo possiamo capire dall'istruzione "jz short loc_40102B" la quale fa un salto condizionale alla posizione di memoria indicata se il risultato della comparazione (cmp [ebp+var_4], 0) è zero, indicando così che la condizione è soddisfatta.

Nel caso in cui la condizione sia soddisfatta il blocco di codice successivo viene eseguito. Questo blocco gestisce il caso in cui la connessione Internet è attiva e stampa un messaggio di successo. In caso contrario, passa al secondo blocco e stampa un messaggio di errore.

In tutti e due i casi il programma esegue la chiusura dello stack per ripristinare lo stato precedente e tornando al chiamante.

3. SIGNIFICATO DELLE SINGOLE RIGHE DI CODICE

push ebp	Salva il valore di "ebp" nello stack.
mov ebp, esp	Imposta "ebp" con il valore corrente dello stack.
push ecx	Salva il valore "ecx" nello stack.
push0; dwReserved	Mette 0 nello stack, cioè il valore "dwReserved" che verrà passato alla funzione "InternetGetConnectedState".
push0; lpdwFlags	Mette un altro 0 nello stack, questa volta il valore del parametro lpdwFlags.
Callds: InternetGetConnectedState	Chiama la funzione.
mov [ebp+var_4], eax	Memorizza il valore di ritorno della funzione "eax" nella variabile locale [ebp+var_4].
cmp [ebp+var_4], 0	Compara il valore memorizzato in [ebp+var_4] con 0.
jz short loc_40102B	Esegue un salto condizionale alla locazione indicata se il confronto precedente è vero.
Push offset aSuccessInterne; "Success:InternetConnection\n	Mette l'indirizzo della stringa "Success: Internet Connection\n" nello stack. Un messaggio di successo che verrà stampato o utilizzato.
call sub_40117F	Chiama la funzione che ha l'indirizzo sub_40117F.
add esp, 4	Aggiunge 4 al registro "esp". Questo è probabilmente un modo per "ripulire" lo stack dopo la chiamata della funzione.
mov eax, 1	Mette il valore 1 nel registro eax.
jmp short loc_40103A	Salto incondizionato (jump) alla locazione loc_40103A.
loc_40102B	inizio di un blocco di istruzioni che gestisce il caso in cui non c'è connessione a Internet.
push offset aError1_NoInte	Mette l'indirizzo della stringa "Error 1.1: No Internet\n" nello stack.
call sub_40117F	Chiama la funzione che ha l'indirizzo sub_40117F.
add esp, 4	Aggiunge 4 al registro "esp". Come detto precedentemente è un modo per "ripulire" lo stack dopo la chiamata della funzione.
xor eax, eax	Imposta "eax" a 0.
loc_40103A	Posizione specifica nel codice.
mov esp, ebp	Muove il valore corrente di 2ebp" nel registro "esp".
pop ebp	Estrae il valore superiore dello stack e lo colloca nel registro ebp. Questa operazione si fa per ripristinare il valore originale di "ebp".
retn	Restituisce il controllo al chiamante.
sub_401000 endp	Indica la fine della procedura.