# PROGETTO S10/L5

di Giuseppe Lupoi

# INDICE

# TRACCIA 1° PARTE

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:

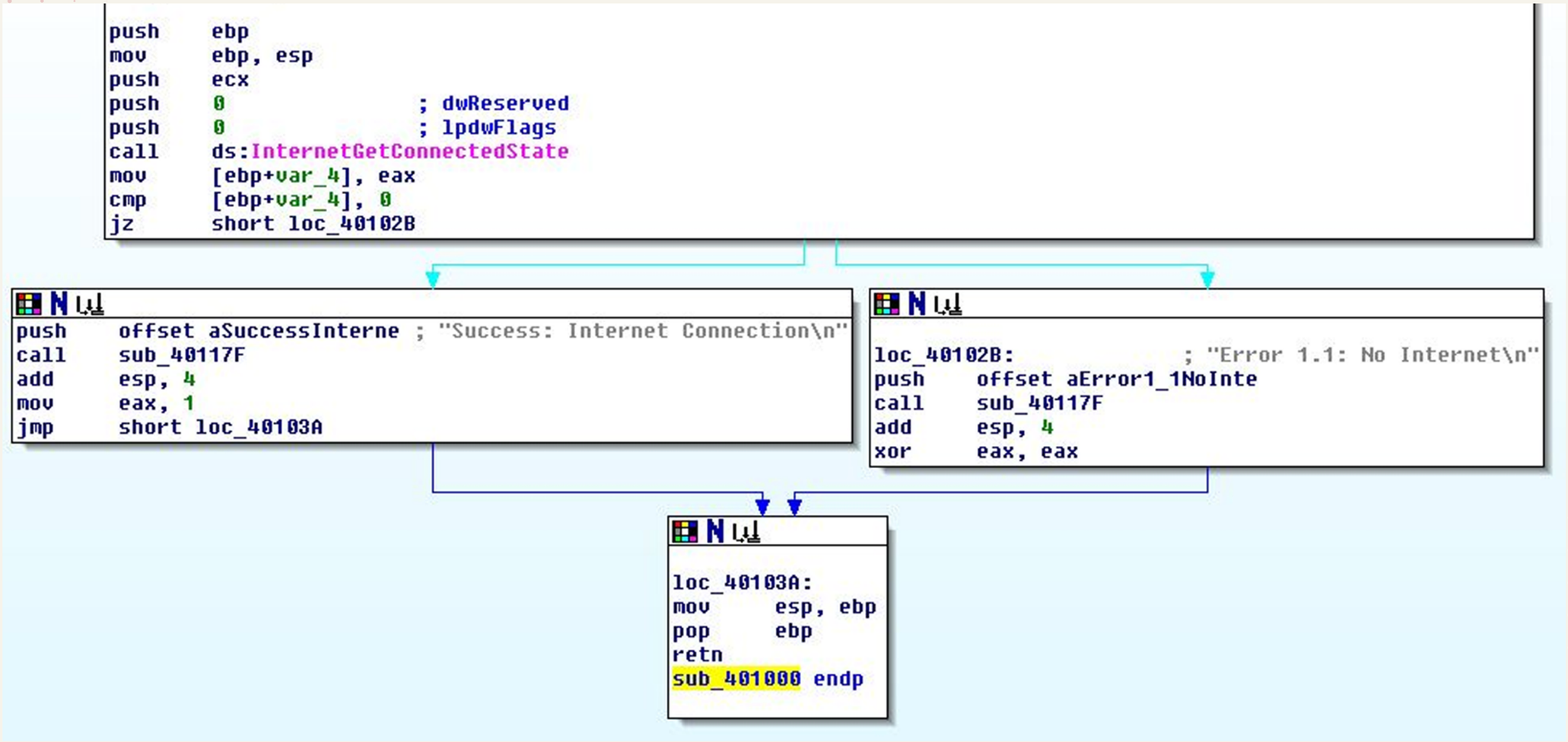
- Quali librerie vengono importate dal file eseguibile?
- Quali sono le sezioni di cui si compone il file eseguibile del malware?

# TRACCIA 2° PARTE

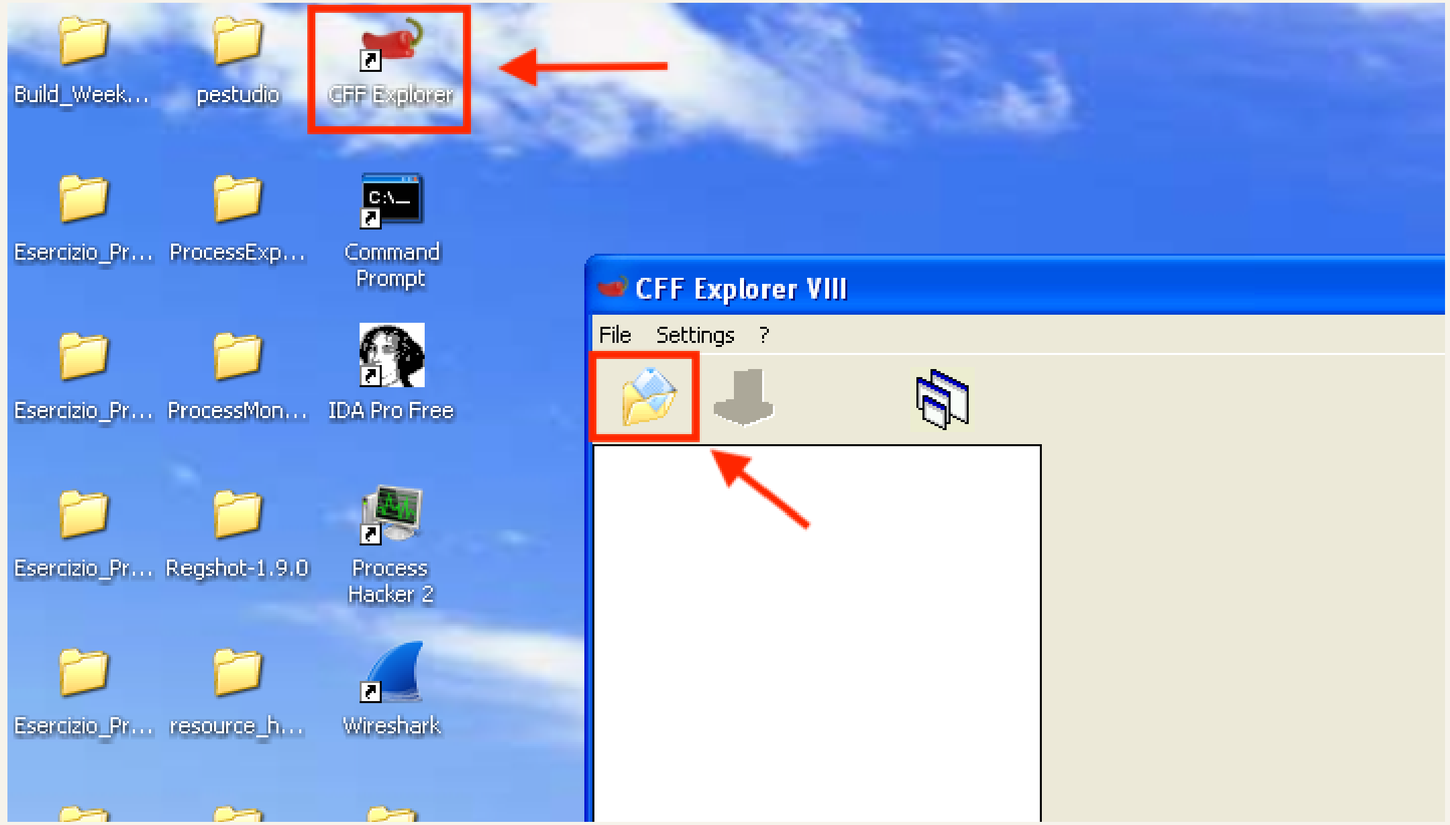
Con riferimento alla figura in slide 3, risponde ai seguenti quesiti:
(Riporto la slide del progetto EPICODE nella prossima slide)

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

# TRACCIA 2° PARTE

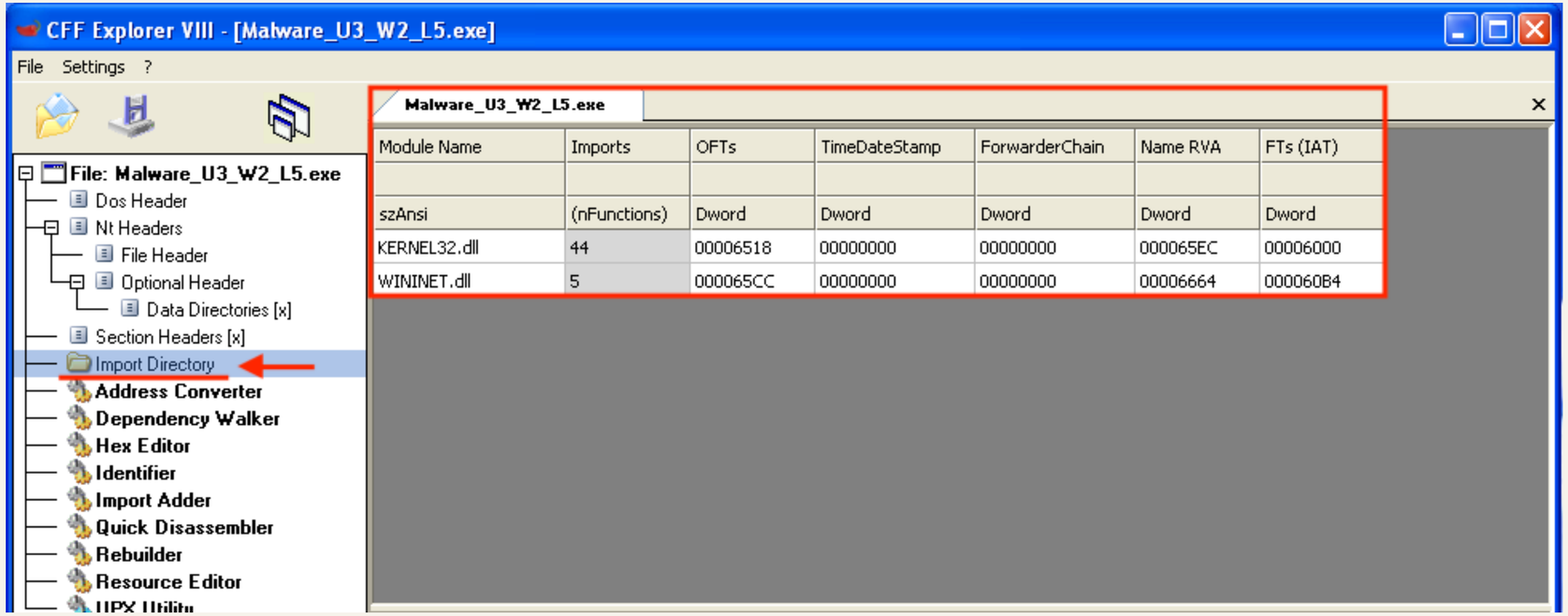


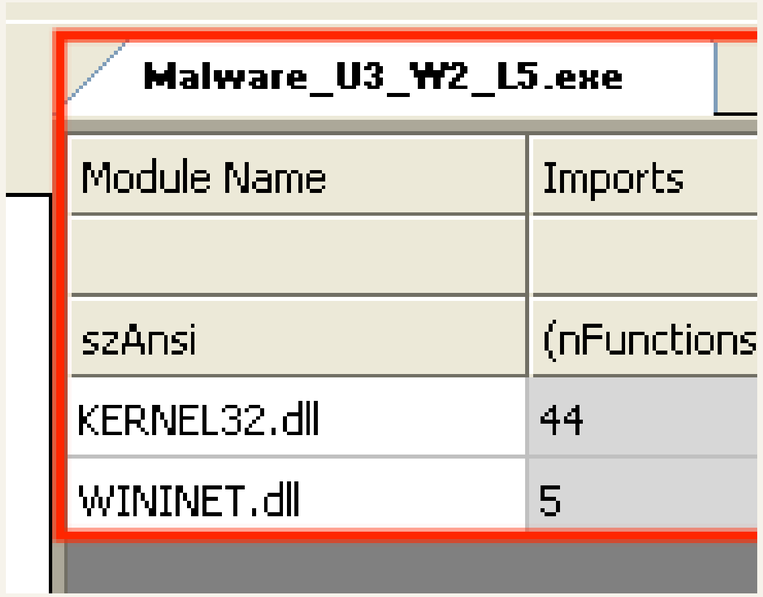
Per lo svolgimento della prima parte della traccia del progetto inizieremo avviando la VM a noi fornita.




Una volta effettuato l'accesso con le credenziali (malware), dal desktop avviamo **CFF Explorer** come indicato dall'immagine (icona con peperoncino rosso), e una volta aperta la schermata del programma navighiamo attraverso la certella indicata dalla freccia per caricare il file di nostro interesse:
Oggi ci occuperemo del **Malware_U3_W2_L5** contenuto in **Esercizio_Pratico_U3_W2_L5**.

Una volta caricato il file **CFF Explorer** si occuperà di scansionarlo per identificare le varie **librerie** e **sezioni** di cui è composto il **malware** in questione.
Come potete vedere dall'immagine sottostante, se tra la voci che vediamo nella colonna di sinistra ci spostiamo su "**Import Directory**", nella schermata di destra ci verrà aperta una griglia contenente le librerie da cui è composto il malware e altri dettagli.





Malware_U3_W2_L5.exe

| Module Name | Imports |
|---|---|
| | |
| szAnsi | (nFunctions |
| KERNEL32.dll | 44 |
| WININET.dll | 5 |

Facendo uno zoom della slide precedente notiamo che sono presenti le seguenti librerie:

- KERNEL32.dll
- WININET.dll

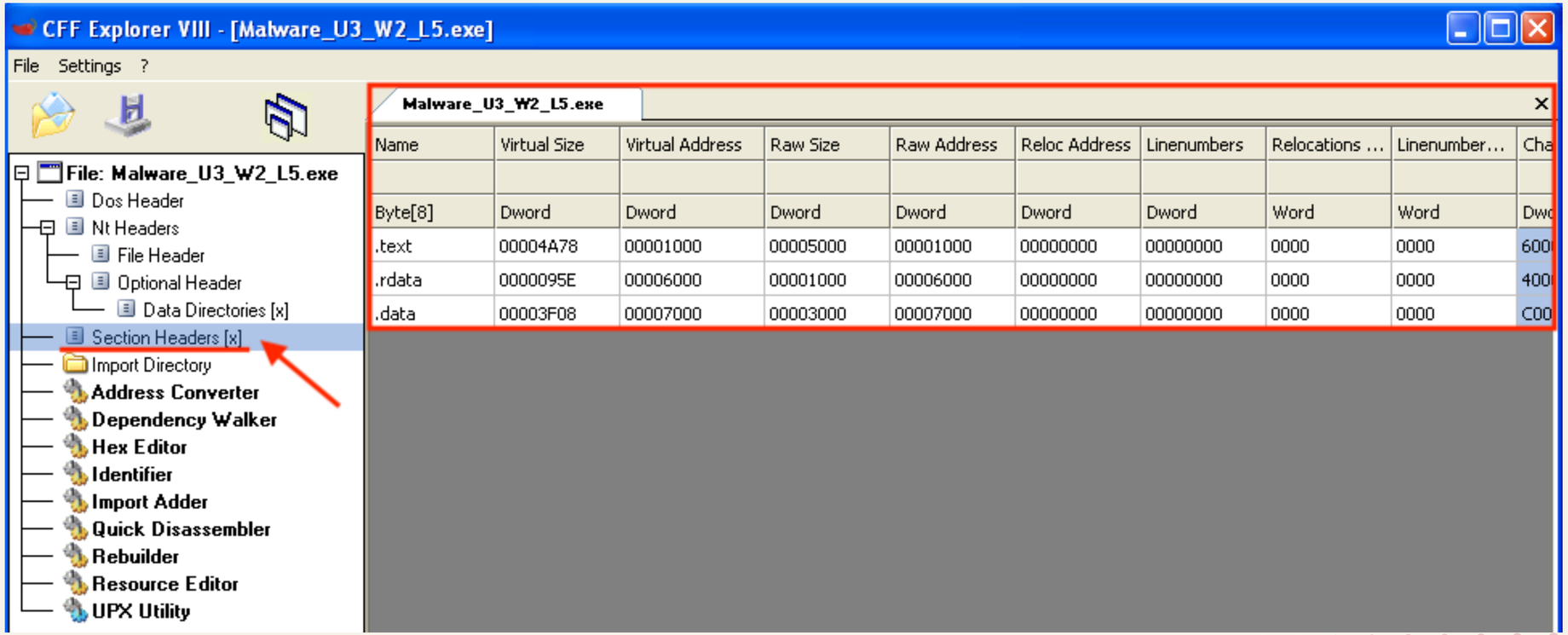
Abbiamo già incontrato le librerie citate sopra nelle lezioni precedenti, quindi ricordiamo che:

- *KERNEL32.dll* è una libreria che contiene le principali funzioni per interagire con il sistema operativo. Dà ad esempio la possibilità di gestire la memoria o manipolare dei file.

- *WININET.dll* invece è una libreria contenente funzioni per l'implementazione di protocolli di rete come HTTP, FTP, NTP.

Una volta individuate le librerie presenti, per svolgere il secondo punto della traccia, individueremo le **sezioni** da cui è composto il malware.
Torniamo nella colonna di sinistra questa volta clicchiamo sulla voce "**Selection Headers [x]**" come indicato dalla freccia, nella schermata presente sulla destra, come in precedenza, ora avremo uno schema delle sezioni presenti nel malware.
Dove possiamo riconoscere:

- .text
- .rdata
- .data




| Name | Virtual Size | Virtual Address | Raw Size | Raw Address | Reloc Address | Linenumbers | Relocations ... | Linenumber... | Cha |
|---|---|---|---|---|---|---|---|---|---|
| Byte[8] | Dword | Dword | Dword | Dword | Dword | Dword | Word | Word | Dwo |
| .text | 00004A78 | 00001000 | 00005000 | 00001000 | 00000000 | 00000000 | 0000 | 0000 | 600 |
| .rdata | 0000095E | 00006000 | 00001000 | 00006000 | 00000000 | 00000000 | 0000 | 0000 | 400 |
| .data | 00003F08 | 00007000 | 00003000 | 00007000 | 00000000 | 00000000 | 0000 | 0000 | C00 |

Approfondiamo le sezioni individuate che abbiamo già visto nelle precedenti lezioni:

- *.text* è la sezione contenente le righe di codice che eseguirà la CPU all'avvio del programma, generalmente questa è l'unica parte che viene eseguita dalla CPU in quanto le altre sezioni possono contenere dati o informazioni di supporto.

- *.rdata* questa sezione contiene generalmente informazioni sulle librerie importate dal file.

- *.data* contiene i dati e le variabili dichiarate globalmente dal programma e che di conseguenza devono essere accessibili da qualsiasi punto.

Per lo svolgimento della 2° parte della traccia invece andremo ad analizzare il codice Assembly a noi fornito che ho riportato in slide n°5.

Proverò ad identificare quindi costrutti a noi noti, creazione degli stack ed eventuali cicli.
Nelle prossime pagine analizzerò il codice per ogni riga da cui è composto.

Riporto quindi le prime 3 righe di codice, dove possiamo notare:

***push*** ***ebp*** ⬅ *la creazione dello stack, in questa riga si salva il registro di base EBP*

***mov*** ***ebp, esp*** ⬅ *questa riga crea un nuovo stack importando il valore di ESP nel registro EBP per la prossima funzione*

***push*** ***ecx*** ⬅ *con questa riga il valore di ECX viene salvato nel registro appena creato*

In queste prime 3 righe abbiamo quindi notato la creazione dello **stack** e dei puntatori di base.

Continuiamo con:

***push*** ***0*** ⬅ in questa riga viene inserito 0 nello stack.
Si tratta di un parametro **dwReserved**

***push*** ***0*** ⬅ in questa riga viene inserito 0 nello stack.
Si tratta di un parametro **lpdwFlags**

***call*** ***ds: InternetGetConnectedState*** ⬅ chiamata alla funzione. Questa particolare funzione verificherà la connessione ipotetica ad internet di una macchina, salvandone poi il risultato in **EAX** precedentemente creato

In queste 3 righe abbiamo quindi notato la l'impostazione degli **stack** a 0 e la chiamata di una funzione.

Continuiamo con:

***mov [ebp, var_4], eax*** ⬅ in questa riga viene salvato il risultato della funzione **InternetGetConnectedState** vista in precedenza

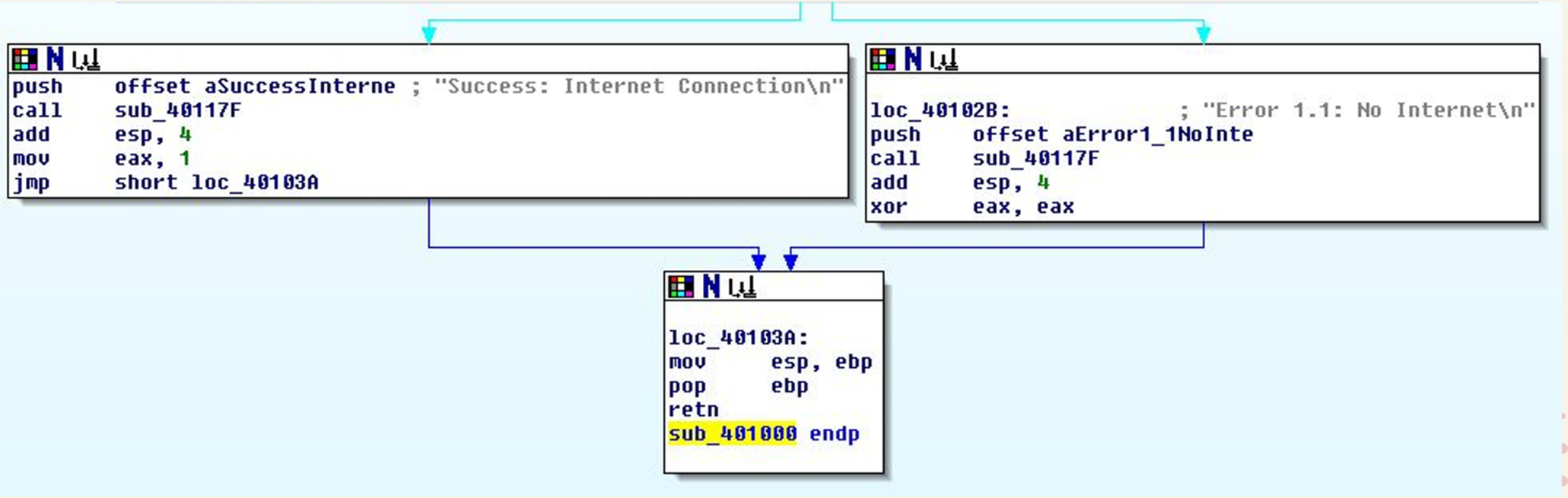
***cmp [ebp, var_4], 0*** ⬅ il suo lavoro è quello di comparare il risultato della funzione con **0** per poi impostare su **0 o 1 i flag Zero e Carry**

***jz short loc_40102B*** ⬅ salto condizionale definito appunto dalla comparazione di **cmp,** in caso di risultato 0 questa riga salta direttamente alla locazione **40102B**

Da queste 3 righe possiamo invece riconoscere un costrutto “**if**” dove appunto in base alla variabile VERA o FALSA il programma prende strade diverse.

Siamo quindi arrivati a questo punto, in base alla comparazione dei flag il programma potrà prendere 2 strade ovvero:

- Risposta positiva, la connessione a internet è presente
- Risposta negativa, non esiste connessione ad internet



Proseguiamo l'analisi del codice ripartendo dallo schema di sinistra visto nella slide precedente, dove leggiamo:

***push offset aSuccessInterne ; "Success: Internet Connection\n"*** ⬅ in questa casistica di successo la connessione esiste e veniamo avvisati con un messaggio a schermo che ci dirà appunto **"Success: Internet Connection\n".**

***call sub_40117F*** ⬅ chiamata a subroutine, il programma si sposterà quindi alla locazione **40117F**

***add esp, 4*** ⬅ in questa riga verrà addizionato il valore **4** a quello presente nel registro **ESP**

***mov eax, 1*** ⬅ qua invece il valore **1** verrà copiato nel registro **EAX** aggiornando quindi il suo valore per indicare il successo alla connessione

***jmp short loc_40103A*** ⬅ salta alla locazione definita per continuare il programma

Nella slide precedente abbiamo analizzato la casistica di successo alla connessione ad internet vedendo come il codice ci mostrerà un messaggio di successo avvisandoci per poi continuare il suo percorso.

Andiamo ad analizzare ora la casistica in cui non vi è una connessione attiva, leggiamo quindi lo schema di destra della slide a pagina n°15.

Nello schema di destra possiamo invece leggere:

***loc_40102B:  ;"Error 1.1: No Internet\n"*** ⬅ in questa casistica dove invece non c'è una connessione ad internet il codice salta alla locazione **10402B** e ci avvisa col messaggio "**Error 1.1: No Internet\n**".

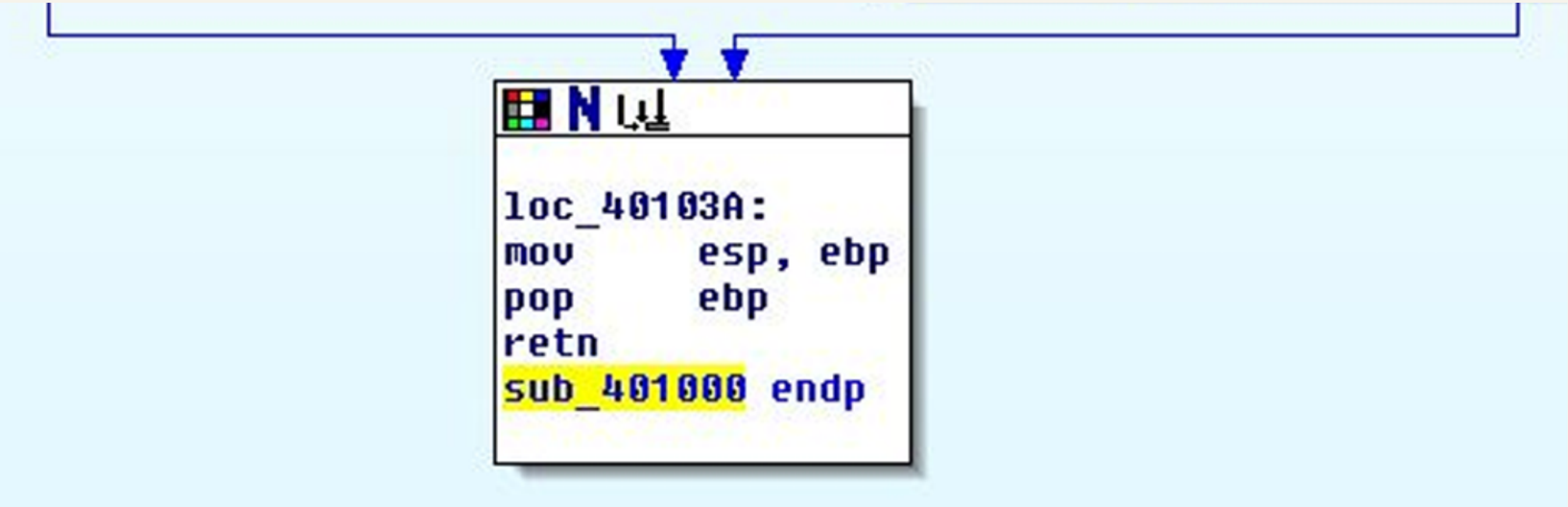
***push  offset a Error1_1NoInte*** ⬅ in questa istruzione viene inserito nello **stack** la stringa alla locazione di "**Error1_1NoInte**" che è appunto correlata al messaggio di connessione non esistente

***call  sub_40117F*** ⬅ chiamata a subroutine, il programma si sposterà quindi alla locazione **40117F**

***add  esp, 4*** ⬅  in questa riga verrà addizionato il valore **4** a quello presente nel registro **ESP**

***xor  eax, eax*** ⬅ l'effetto di questa operazione è riportare i valori del **registro EAX a 0**. Questa istruzione è spesso utilizzata per azzerare un registro prima di utilizzarlo per un nuovo calcolo o per inizializzare una variabile a 0.

Abbiamo a questo punto analizzato anche la casistica dove dopo aver controllato il codice non individua una connessione ad internet sulla macchina.
Non ci resta che analizzare l'ultima parte del codice a noi fornito, che riporterò in calce.



In questa ultima parte leggiamo quello che è il blocco di istruzioni che rappresenta la fine del codice:

***loc_40103A:*** ⬅ in questa riga il codice si sposterà alla locazione definita per terminare i suoi compiti

***mov esp, ebp*** ⬅ questa istruzione sposta il valore contenuto nel registro **EBP** nel registro **ESP**. Questa operazione viene eseguita prima di tornare da una subroutine per ripristinare lo stack pointer allo stato in cui era prima che la subroutine fosse chiamata

***pop ebp*** ⬅ con questa istruzione si rimuove lo stack EBP per riportarlo al suo valore iniziale

***retn*** ⬅ in questa riga si esegue un ritorno alla subroutine iniziale

***sub_401000 endp*** ⬅ in questo caso “**endp**” indica la fine della procedura quindi il programma tornerà alla locazione definita cioè **401000**

Al termine dell’analisi del codice Assembly che abbiamo visto possiamo sicuramente riconoscere la creazione dello **stack** vista all’inizio, il costrutto “**if**” che permette al programma di prendere decisioni in base alla variabile se risulta 0 o 1 ovvero VERO o FALSO.

Dopo queste valutazioni possiamo dire che il codice che abbiamo visto fin’ora con la funzione “**InternetGetConnectedState**” serve appunto per controllare l’eventuale connessione ad intenet di una macchina.