Analisi statica di malware

CFF Explorer, Assembly
Alessandro Morabito @ Epicode

1 Introduzione

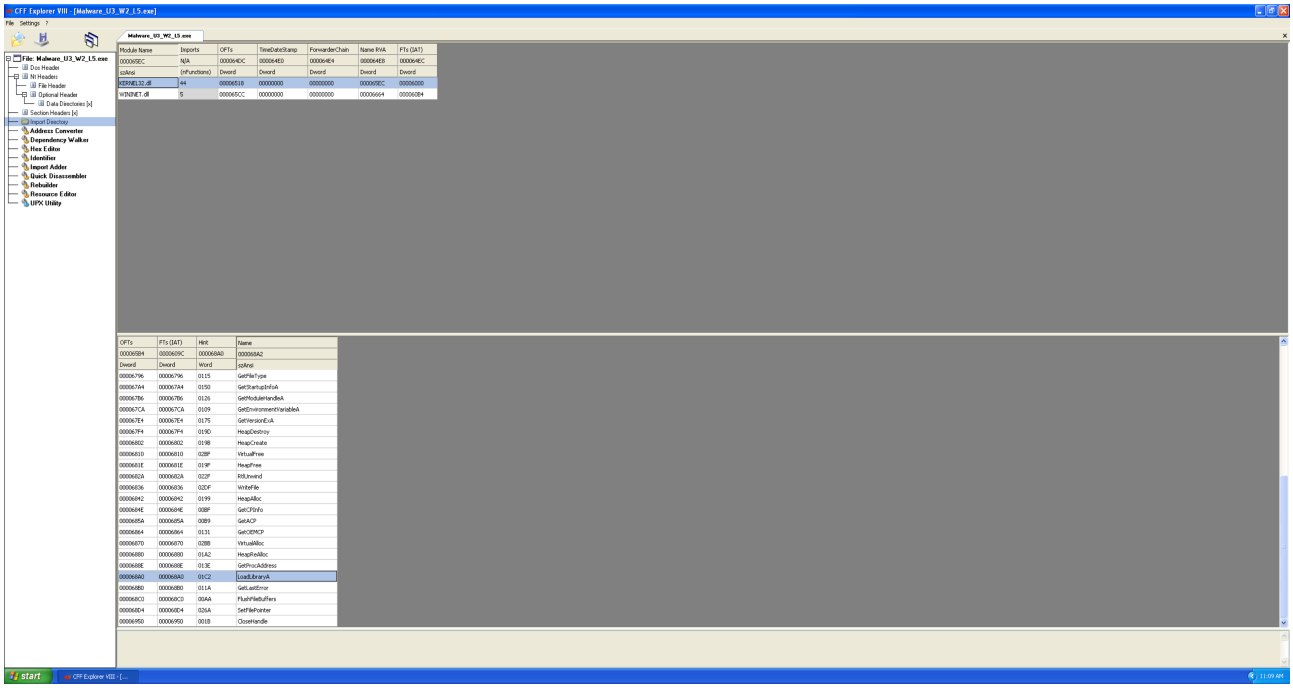
Questo progetto è suddiviso in due sezioni principali: nella prima ci viene chiesto di effettuare l'analisi statica basica di un malware, indicando le librerie importate e le sezioni del file eseguibile; nella seconda, apparentemente scollegata dalla prima, ci viene chiesto di commentare il codice *assembly* di un malware. Questo corrisponde ad effettuare una analisi statica avanzata di una sezione di un malware.

2 Analisi statica basica

2.1 Librerie

2.1.1 `KERNEL32.dll`

La libreria `KERNEL32.dll` è una *Dynamic Linked Library* fondamentale che permette, ad esempio, di effettuare operazioni sulla memoria e avviare processi. Nel nostro caso vediamo che sono state importate 44 funzioni. Tra queste ci soffermiamo in particolare su `WriteFile`, che permette la scrittura, quindi la creazione, di un file, e su `LoadLibrary` (evidenziata in figura), che permette di caricare in memoria un modulo. Inoltre `GetProcAddress` (la riga sopra rispetto a quella evidenziata nell'immagine) permette di ottenere l'indirizzo verso una funzione che può essere stata importata con la funzione precedente.
Queste ultime due funzioni di `KERNEL32.dll` permettono quindi di caricare una libreria in *runtime*, il che impedisce all'analisi statica basica di ottenere tutte le informazioni circa le librerie caricate.



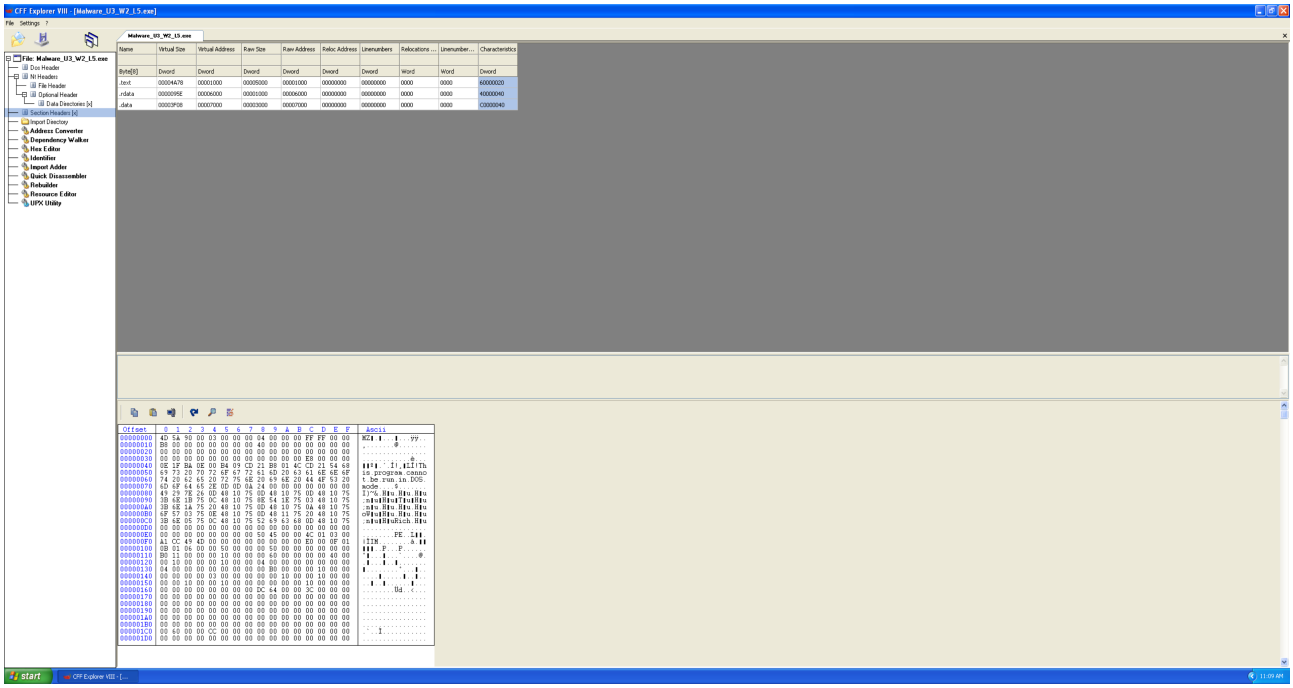
2.1.2 WININET.dll

La libreria WININET.dll mette a disposizione funzioni che permettono l'utilizzo di protocolli quali FTP e HTTP. In questo malware sono importate 5 funzioni. Tra queste notiamo InternetOpenURL, che permette di aprire una risorsa ad un certo URL con protocollo FTP o HTTP, e InternetReadFile, che permette di leggere i dati presenti all'URL indicato dalla funzione precedente.

2.2 Sezioni

All'interno di questo malware possiamo individuare 3 diverse sezioni:

- .text , contenente le istruzioni da eseguire;
- .rdata , contenente informazioni riguardanti librerie importate/esportate (più in particolare, dati di sola lettura. Le informazioni riguardanti librerie importate ed esportate possono altrimenti trovarsi all'interno di .idata e .edata rispettivamente);
- .data , contenente variabili globali.



3 Analisi statica avanzata

Ci viene fornito un codice assembly ottenuto con un disassemblatore. Ci viene chiesto di analizzarlo individuando vari costrutti e di ipotizzare il comportamento generale di tale codice.

```
push ebp
mov ebp, esp
```

Questo costrutto prepara lo stack per una chiamata di funzione.
Il registro ebp, *Extended Base Pointer*, punta alla base dello stack. Il motivo per cui viene effettuato un push del contenuto del registro sullo stack è che, dopo che una funzione ha terminato le sue istruzioni, il codice eseguito successivamente sarà quello della funzione chiamante. Salvando in maniera efficiente la base dello stack della funzione chiamante è quindi possibile ricostruire il suo

stack al termine della funzione chiamata.

```
push    ecx
push    0                                       ; dwFlags
push    0                                       ; lpdwFlags
call    ds:InternetGetConnectedState
mov     [ebp+var_4], eax
```

In questa porzione di codice i comandi `push` vengono utilizzati per passare i parametri alla funzione che verrà successivamente chiamata con il comando `call`. Quest'ultima istruzione modifica il valore del registro `eip` (*Extended Instruction Pointer*), che punta all'indirizzo della prossima istruzione da eseguire, dopo averne salvato il valore sullo stack. Il valore originale del registro `eip` permetterà di riprendere il flusso del programma al termine della funzione tramite il comando `ret`, che rimuove il valore dallo stack assegnandolo al registro `eip`.
Infine, il comando `mov` assegna il valore di `eax` a una variabile, indicata usando come punto di riferimento l'indirizzo della base dello stack, ovvero il registro `ebp` (*Extended Base Pointer*). Il registro `eax` viene utilizzato per salvare il valore di ritorno della funzione.
`InternetGetConnectedState` è una funzione che verifica se è presente una connessione a un modem o una LAN. Successivamente, ritorna `TRUE` (`1`) se essa è presente, `FALSE` (`0`) altrimenti.
In questa sezione di codice, quindi, viene effettuata una chiamata alla funzione `InternetGetConnectedState` prima, e successivamente il valore di ritorno viene assegnato ad una variabile locale della funzione chiamante. Inoltre, notando che non viene effettuata alcuna operazione sugli estremi dello stack immediatamente dopo la chiamata alla funzione, possiamo dedurre che essa venga chiamata con metodologia `STDCALL`: la stessa funzione chiamata si occupa di eliminare il proprio stack prima di tornare alla funzione chiamante.

```
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

Questa porzione di codice corrisponde ad un costrutto `if`. La prima riga confronta il valore della variabile locale (quindi, in questo caso, il valore di ritorno della funzione `InternetGetConnectedState`) con il valore `0`. Questo avviene prima effettuando una operazione `sub` di sottrazione tra la prima e la seconda variabile, senza andare a modificare il valore della prima. Se il risultato di tale operazione è nullo, allora la `ZF` (*Zero Flag*) assume valore unitario. Se invece esso è positivo, allora la `CF` (*Carry Flag*) risulta nulla. In caso contrario, `CF` assume valore unitario.
L'istruzione `jz` è un salto condizionale che viene effettuato se il valore di `ZF` è unitario, quindi se il valore di ritorno della funzione `InternetGetConnectedState` è nullo, ovvero la macchina non è connessa a modem o LAN. In tal caso il salto viene effettuato verso la posizione indicata dal *label* `loc_40102B`. In caso contrario, l'istruzione eseguita sarà quella immediatamente successiva.

```
push    offset aSuccessInterne      ; "Success: Internet Connection\n"
call    sub_40117F
add     esp, 4
mov     eax, 1
jmp     short loc_40103A
```

Questa porzione di codice è una chiamata di funzione. A differenza di quella vista sopra, tuttavia, si nota come subito dopo il suo ritorno viene eseguita un'operazione `add` di addizione sul registro `esp`, che punta alla cima dello stack (metodologia `cdecl`). `esp` aumenta di `4 byte`, restringendo quindi lo stack ed eliminando il parametro passato alla funzione appena chiamata. Quest'ultimo è un riferimento alla stringa `aSuccessInterne`, quindi un indirizzo a `32 bit` ovvero `4 byte`. L'istruzione `mov` viene utilizzata per assengare il valore `1` al registro `eax`. Infine, viene effettuato un salto verso `loc_40103A` per evitare di eseguire il codice immediatamente successivo, ovvero:

```
loc_40102B:
push    offset aError1_1NoInte      ; "Error 1.1: No Internet\n"
call    sub_40117F
add     esp, 4
xor     eax, eax
```

Mentre la prima riga non è un'istruzione, bensì un label per l'indirizzo di memoria corrispondente all'istruzione `push` successiva, anche questo estratto corrisponde a una chiamata di funzione con metodologia `cdecl`. Al termine della funzione chiamata viene quindi ristretto lo stack operando sul registro `esp`. Successivamente, con l'istruzione `xor` sul registro `eax` con sé stesso, esso viene annullato.

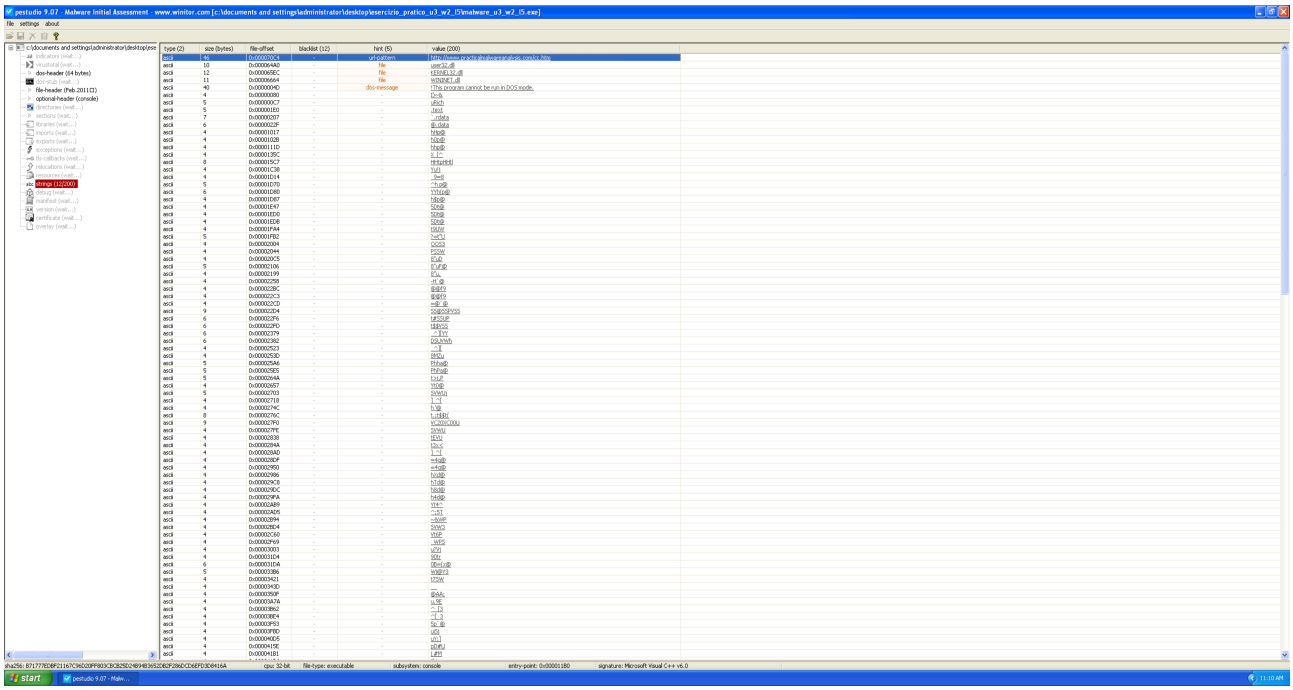
```
loc_40103A:
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

Possiamo vedere qua come avviene la pulizia dello stack al termine di una funzione, in accordo con la metodologia `STDCALL`: inizialmente il puntatore alla cima dello stack assume il valore di quello alla base dello stack della funzione chiamata. Successivamente il valore in cima allo stack, corrispondente all'indirizzo della base dello stack della funzione chiamante, viene assegnato a `ebp` e rimosso dallo stack con l'istruzione `pop`, che restringe lo stack andando a modificare il valore di `esp`. Infine, con l'istruzione `retn` viene effettuata un'operazione analoga al `pop`, rimuovendo l'indirizzo in cima allo stack, salvato con il comando `call`, e assegnandolo al registro `eip`, che punterà all'istruzione successiva della funzione chiamante (non presente nell'estratto). L'ultima riga di codice indica il termine della funzione.
Quando la funzione chiamante riprenderà l'esecuzione, al registro `eax` sarà assegnato il valore di ritorno della funzione appena analizzata. Questo avrà quindi un valore `1` in caso la macchina sia connessa alla rete (`mov eax, 1` prima del salto incondizionato verso `loc_40103A`), `0` altrimenti (`xor eax, eax`).

3.1 Conclusioni

Dopo aver effettuato una analisi statica basica su un malware e una analisi statica avanzata su un estratto di codice ottenuto con un disassemblatore, si possono notare similarità tra i due programmi. Innanzitutto si nota che, così come il malware effettua una importazione dinamica della libreria `WININET.dll`, il codice fornitoci utilizza anch'esso delle funzioni (che possiamo vedere importate con `CFF Explorer`) che fanno riferimento a una connessione ad internet. Ciò nonostante, non si può

trarre la conclusione che il codice fornito sia un estratto del malware analizzato, in quanto l'azione di controllare la connessione ad internet è molto generica e comune.
Utilizzando il comando `strings` sull'eseguibile o il programma `PE Studio`, possiamo individuare un URL: `http://www.practicalmalwareanalysis.com/cc.htm`.



Considerando che tra le funzioni importate si è trovato `InternetReadFile` dalla libreria `WININET.dll` e `WriteFile` dalla libreria `KERNEL32.dll`, posso dedurre che il malware in questione sia un *downloader*, che si connette all'`URL` indicato precedentemente, legga con la funzione `InternetReadFile` un altro malware associato, quindi lo scriva su disco con `WriteFile`.