Lezione 12 ottobre 2023

Primi passi con Ghidra

Analisi di un file a 32 bit Windows. Il formato è di tipo Portable Executable (PE), usato per file eseguibili, file oggetto, librerie condivise e device drivers, Questo formato presenta delle *testate*. La prima testata è **IMAGE_DOS**, ma al giorno d'oggi è inutile, è un retaggio del passato. Successivamente troviamo **IMAGE_NT_WIN_32:**, questa testata è espandibile con Ghidra per vederne il contenuto (c'è il tasto + sulla sinistra).
E' possibile vedere informazioni come *sezioni*, *timestamp* e **Optional Header**, che in realtà, NON è opzionale come si potrebbe pensare, anzi è fondamentale oggi!
Osservazione: Se leggo un file dal SO, viene aggiornato il metadato associato all'ultima apertura. Questa informazione non è del file, ma del sistema operativo. L'eseguibile analizzato ha tutte le info necessarie per creare il processo, come la zona del codice, dei dati etc... che non sono metadati. **AddressEntryPoint**: 4 byte, non ha indirizzi bensì **RVA**, cioè offset in riferimento a dove il sistema operativo va a caricare il codice, che è la *base of code*.

Sappiamo che i file eseguibili sono rilocabili, posso caricarli in posizioni diverse da dove sono stati idealmente pensati, e questo richiede di modificare anche indirizzi associati, come se stessimo traslando. Le *posizioni relative* restano immutate, se una istruzione dista 5 linee da un'altra, non mi importa dove viene posto il blocco, sempre 5 linee le separano. Ricordiamo che disassemblatore $\neq$ debugger, poichè noi non stiamo eseguendo nulla, vedo solo dove il compilatore ha predisposto le componenti.

Analisi dei tipi di indirizzi

- **RVA**: è un offset rispetto ad un indirizzo di base in cui viene caricata una certa sezione.
- L'**offset** è uno spazzamento rispetto al file. Questi aspetti li troviamo nel file eseguibile.
- Quando eseguiamo `call func1` usiamo componenti $< c_i, v >$ dove v è una parte dell'indirizzo logico.

Noi abbiamo tre tipi di indirizzi:
$\circ \rightarrow |segment| \rightarrow \circ \rightarrow |pagination| \rightarrow \circ$

Noi ci troviamo nel primo cerchio, il secondo cerchio è un *indirizzo virtuale/lineare*. Nel primo cerchio l'indirizzo non è completo. L'indirizzo logico è composto da *(segmento, offset rispetto al segmento)*, noi vediamo solo l'offset v, quindi dobbiamo associargli il segmento. Quale? CS. Questo indirizzo logico *non è* quello del processo.

Cosa è un indirizzo lineare (o virtuale, a livello hardware)?

E' un numero a 32 bit (se il file è a 32 bit), è una cella di memoria che diventa fisica se passa per la *paginazione*. FIle eseguibili possono avere stessi address ed entry point, ma a seconda del processo (o dei flussi dell'eseguibile), posso avere indirizzi lineari uguali a cui associo, tramite paginazione, posizioni diverse per ogni processo. Se i segmenti partono da 0, allora $0 + \ offset = offset = indirizzo\ lineare$ Nei modelli flat (in cui si parte appunto da 0) ho dei segmenti lineari e non logici, in quanto dispongo del dato completo. Come lo capisco? se vedo segmenti *CS,DS,SS*, in quanto nel modello piatto sono caricati a 0.

Aspetti dello stack

Nel file analizzato, c'è l'operazione `sub esp, 0x1c` che può essere "tradotta" da Ghidra come `sub esp,28` mediante il tasto "convert". Lo stack intel dispone di un registro *esp* a 32 bit/4 byte nella versione *x_86* e a 64 bit/8 byte in *x_86_64*. Qui parliamo di memoria lineare, non fisica. Il registro *esp* tiene l'ultimo indirizzo più in alto, cioè l'ultimo scritto. Più lo stack cresce verso l'alto, più gli indirizzi che uso sono piccoli.

```
| ....  |
| code  |
| data  |
| heap  |
| stack |
```

Tra *heap* e *stack* non c'è una divisione netta. Questo perchè:

- L'heap cresce dall'alto verso il basso, più gli indirizzi crescono scendendo.
- Lo stack cresce dal basso verso l'alto, e mentre cresce gli indirizzi diventano più piccoli.

Così entrambi crescono fino alla collisione, senza avere necessità di definire un confine.
Esempio mio: E' come un quaderno per due materie, che usiamo sulle due facciate. La prima materia è come l'heap, parto da indirizzo piccolo (pagina 0) e cresce di pagine. L'altra materia è come stack, parto da indirizzo grande (pagina 100, fine quaderno), e andando indietro "decresco", così la materia che prende più spazio ha la meglio, e non devo però definire una linea di divisione.

Ritorniamo a `sub esp,28`: poichè operiamo a 32 bit, ovvero 4 byte, allora $\frac{28}{4} = 7$, cioè stiamo *allocando 7 posizioni sullo stack*. Questo è un altro indizio che ci suggerisce che il file sia per Windows a 32 bit: Se fosse stato a 64 bit, non potevamo ottenere un numero intero facendo $\frac{28}{8}$, quindi non potevo prendere "bene" lo stack.

Esaminiamo anche `dword ptr [esp] => local_1c, 0x1`: => è realizzato da Ghidra, non è assebler!
Stiamo scrivendo 1 su una cella di memoria, di 4 byte perchè è una *dword*. Poichè lavoriamo in Little Endian, abbiamo $|00|00|00|01$ (infatti, come abbiamo visto nelle vecchie lezioni, possiamo comprimere i 4 byte in 2 byte, che vanno da 00 ad *ff* !) **NB:** Possiamo assegnare, tramite Ghidra, dei nomi o *etichette* per rendere l'analisi più leggibile, oltre a dei commenti.

Altro sullo Stack

Se chiamiamo una funzione, dobbiamo salvare l'indirizzo di ritorno, usando proprio lo stack. Le operazioni sono:

- PUSH: push from instruction pointer, realizzato quando chiamo le call.
- POP: pop to instruction pointer, realizzato quando eseguo una return.

Presa una funzione:

```
int f(int a1, int a2, int a3)
{
 int v;
 int v2;
 int v3;
}
```

v è una variabile automatica persistente/visibile *solo* nella funzione, grazie allo stack. Tuttavia abbiamo detto che lo stack cresce e decresce, il compiler come fa a risalire alla posizione di **v**? Sicuramente devo usare un *riferimento* rispetto **esp**, ma abbiamo detto che **esp** mantiene la cima dello stack, che appunto si muove.

- Dovrei mantenere spaziamenti aggiornati per trovare ogni volta ciò che mi serve, ma sarebbe complesso.
- Posso dedicare un registro base **ebp** come riferimento fisso, l'aspetto negativo è che sto sprecando un registro! Tuttavia questo è il metodo più diffuso.

Nello stack, basandoci sul codice di riferimento sopra, verranno aggiunti nello stack *prima i parametri* (partendo dall'ultimo verso il primo), poi il *return address* e poi *le variabili automatiche*, sempre in ordine inverso perchè lo stack è descending. Avremmo:

```
|    v     |
|   v2     |
|   v3     |
| ret addr |
|   a1     |
|   a2     |
|   a3     |
```

In realtà, sopra *ret addr* andrebbe allocato spazio per il registro *ebp*, fisso. Quindi, partendo dal basso e salendo, avremo ad un certo punto uno spazio per *ebp* che coinciderà proprio con *esp*, e sopra di lui le variabili v_i. In questo modo, sappiamo dove sono collocate rispetto *ebp*.

```
|    v     | <- ebp -12
|   v2     | <- ebp -8
|   v3     | <- ebp -4
|   ebp    | <- esp = ebp
| ret addr | <- ebp +4
|   a1     | <- ebp +8
|   a2     | <- ebp +12
|   a3     | <- ebp +16
```

A livello assembly, ciò che facciamo viene spesso racchiuso dal nome **enter**:

```
push ebp        <- metto ebp nello stack
mov esp, ebp    <- posiziono stack su ebp
sub esp, 12     <- alloco 3 posizioni per le variabili
```

Tuttavia difficilmente troveremo **enter** o **leave** nei nostri malware:

```
mov esp, ebp    <- riporta stack su ebp
pop ebp         <- togliamo ebp, quindi andiamo su ret addr
ret             <- ritorniamo all'address specificato
```

Vedremo poi, se una funzione fa return:

- Se possiede un registro, quel registro di ritorno è *eax*
- Se *non* possiede un registro, fa return di una struttura.