# Contents

# Appunti di Malware Analysis

# 3 ottobre 2023

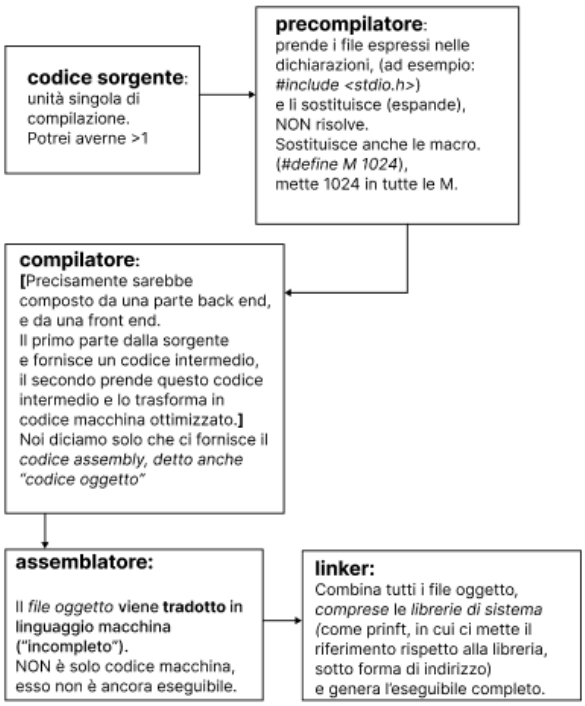
## Sistemi operativi consigliati

Poichè la maggior parte dei malware da analizzare sono sviluppati per Windows, ci serve un sistema Windows per testarli. Questo sistema deve però girare su macchina virtuale, non sul computer host, in quanto una gestione non attenta del malware potrebbe portarlo ad operare sul nostro pc di riferimento. Per evitare ciò, sarebbe buona norma avere un sistema operativo diverso da Windows sul nostro pc host.

## Virtualizzazione

Per creare macchine virtuali, oltre ai soliti VmWare e VirtualBox, il professore suggerisce Qemu, in quanto è un servizio di paravirtualizzazione efficiente, mentre i primi due sono solo emulazioni software. Qemu ha un apprendimento più ripido, ma propone una efficienza migliore. Non è obbligatorio usare Qemu, in quanto la caratteristica principale che vogliamo sono gli snapshot, ovvero la possibilità di generare delle istantanee del sistema. Ad esempio, possiamo fare uno snapshot prima e dopo l'inserimento del malware, e vedere cosa è cambiato.

## Ottenere il codice macchina: processo

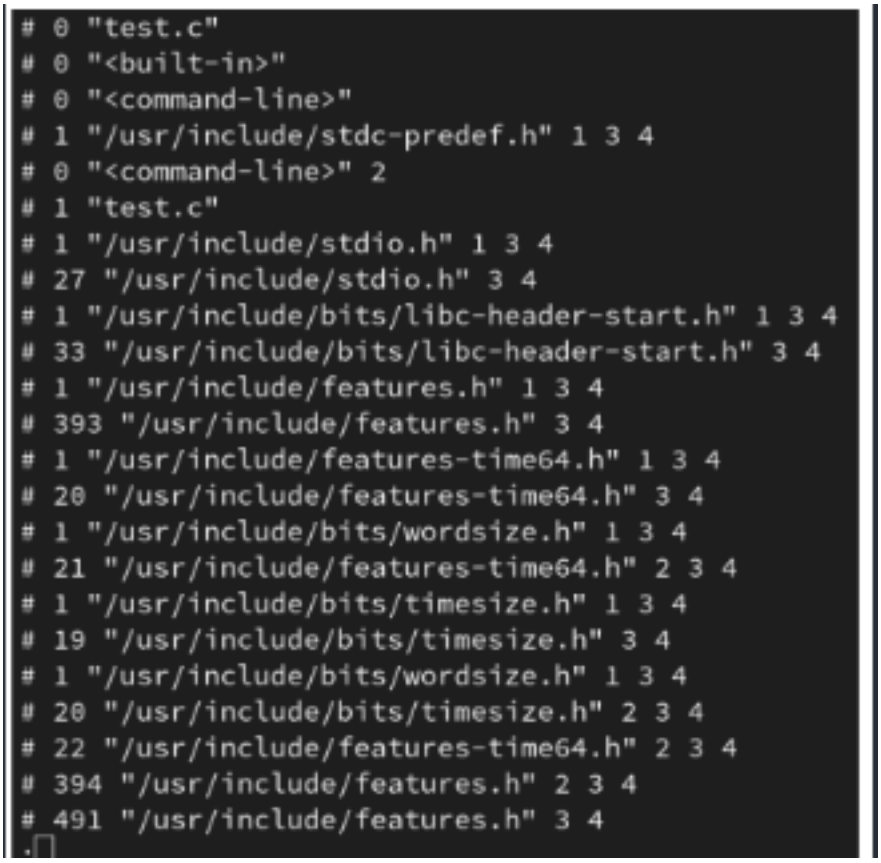



## Flag da GCC

Prendiamo il seguente file che stampa "Hello World":

```c
#include <stdio.h>
int main() {
    // printf() displays the string
    printf("Hello, World!");
    return 0;
}
```

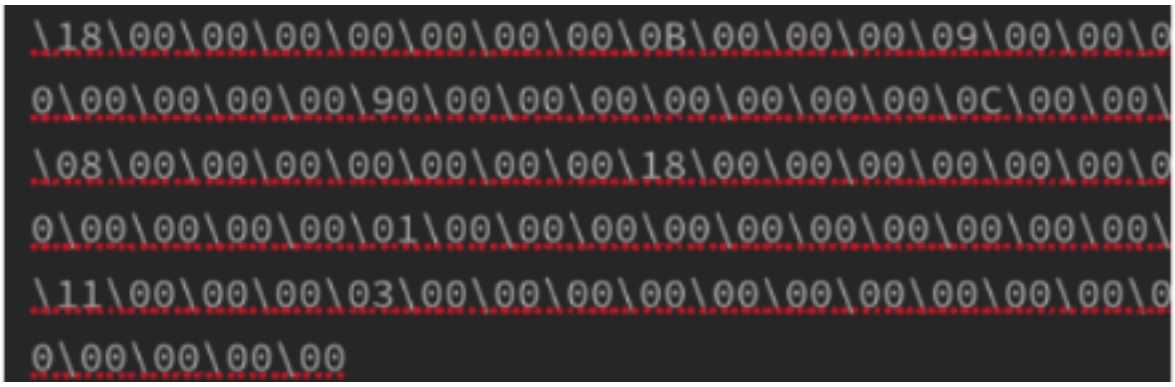
Se compiliamo con `gcc -E test.c|less` otteniamo la sua precompilazione, questa è solo una parte, il codice includerebbe anche prototipi, struct, etc...



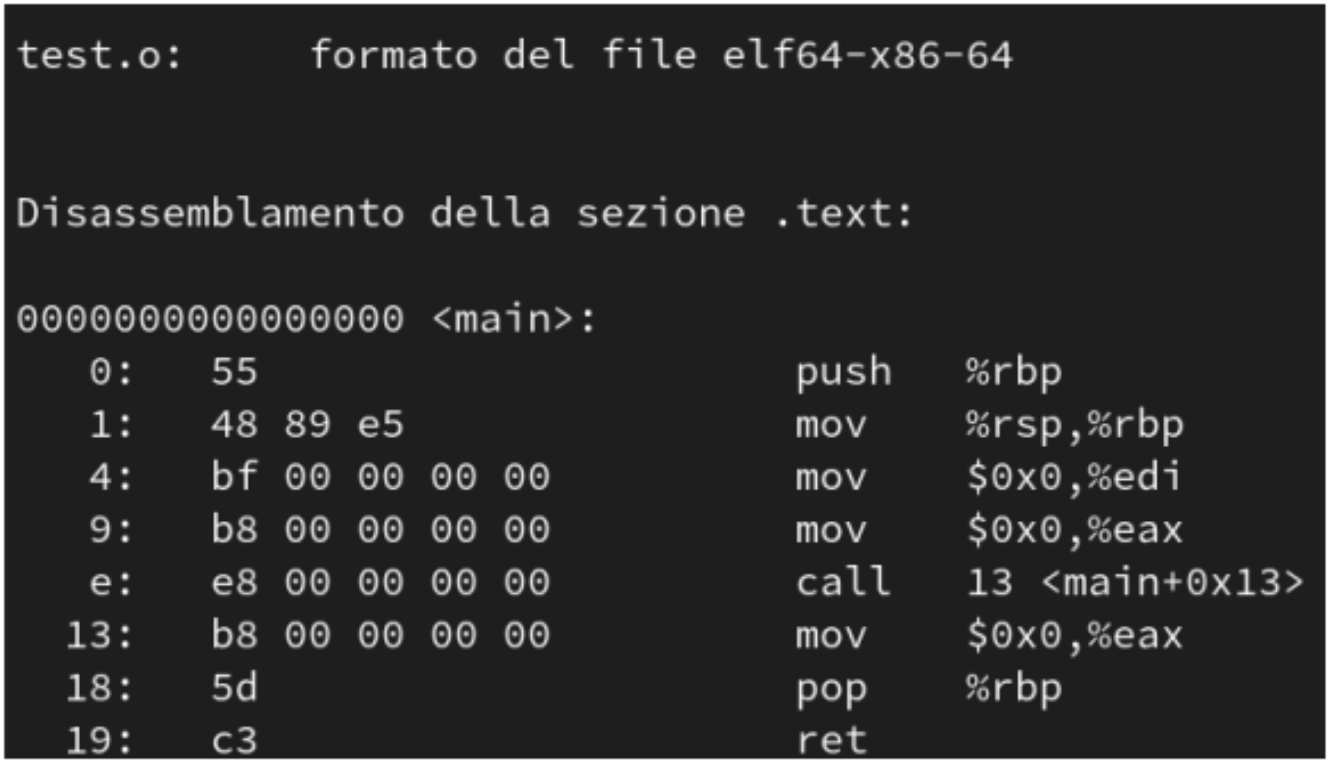

Se compiliamo con `gcc -S test.c -o test.s` otteniamo il suo assembler:

```
        .file   "test.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Infine, con `gcc -c test.c -o test.o` otteniamo le istruzioni macchina




Possiamo usare `objdump -d test.o`per avere più ordine:




## Librerie

### Librerie statiche

Se nel file oggetto esiste un riferimento a printf, il linker mette nel programma eseguibile tutte le istruzioni macchina da printf. Vengono incorporate direttamente nell'eseguibile del programma durante la compilazione. Sostituisco le call prinft con l'indirizzo della procedura. Il codice è nel file eseguibile, libreria non serve più. Opero a compile time.

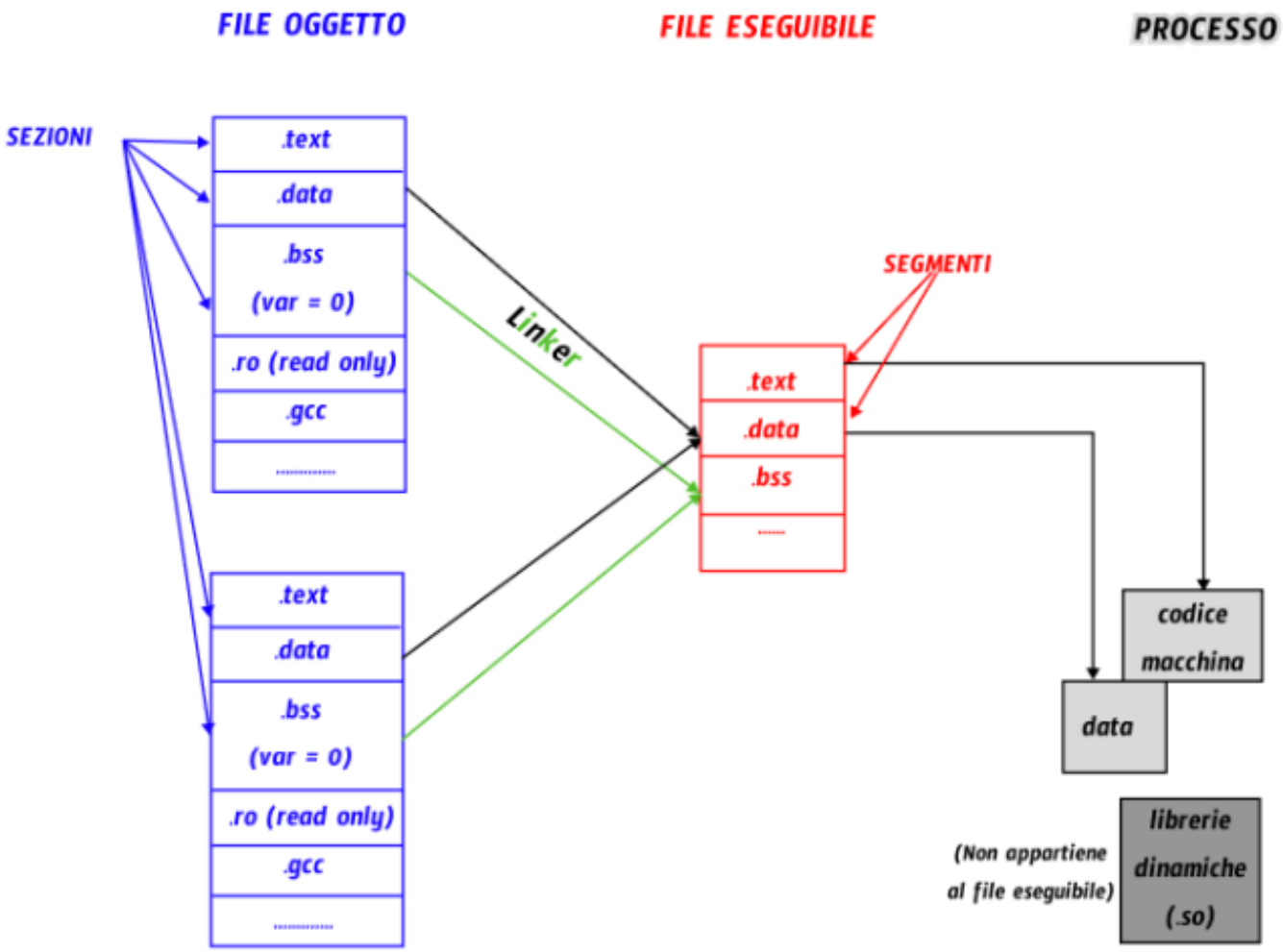
### Librerie dinamiche

Il loro codice non viene incorporato nell'eseguibile, ma avviene un collegamento dinamico a runtime. Il linking avviene col sistema operativo, che mantiene queste librerie dinamiche `.dll` o `.so`. Le dimensioni del file sono minori, in quanto nel programma non ho copiato nulla, ma solo collegato.

**Esempio** Se rimuovessi `stdio.h`, ma il mio programma usa le print, ho un errore già a livello assembly. Questo perchè il precompilatore mette il riferimento, per poi delegare il resto al linker.

## Osservazioni sui file eseguibili

Il file eseguibile NON è un file oggetto. Il file eseguibile contiene tutto il necessario per eseguire. Posso avere anche più file oggetto. Dal file oggetto costruisco un file eseguibile su disco che diventa poi un processo in RAM. Un processo è una istanza di programma in esecuzione, un insieme di informazioni che consentono al programma di tracciare la sua esecuzione.




I file oggetto (in blu) sono composti da sezioni. Il linker lega tutte le sezioni di uno stesso tipo (es: `.text`) in un unico **segmento** (es: `.text`), appartenente al *file eseguibile (in rosso)*. **Sezioni** $\neq$ **Segmenti** (Possiamo trovare *più sezioni* rispetto al numero di *segmenti*.) Passando al **processo**, questo aggiunge elementi non appartenenti al file eseguibile (come le librerie dinamiche.) Posso decompilare un file eseguibile, ma non un file oggetto. Questo perchè un file eseguibile contiene già il codice macchina tradotto in binari eseguibili, ovvero è più vicino all'hardware. Il file oggetto, invece, non è ancora completo, poichè dovrà essere combinato con altri file, essendo solo una parte. Inoltre, è possibile incorrere in ottimizzazioni.

# 5 ottobre 2023

## Tipi di disassemblatori

Il disassemblatore è un tool che legge codice macchina (espresso in byte) e associa, per ogni sequenza di byte, un'istruzione macchina simbolica.

- **Lineari**: Dati dei valori numerici $v_1, ..., v_n$, cioè un flusso di byte, va ad associare ad ogni byte una istruzione macchina (nb: una istruzione macchina potrebbe richiedere più di un byte). Non è detto che tutte le istruzioni abbiano la stessa lunghezza (questo è vero in architetture RISC come ARM, dove 32 bit corrispondono a 4 byte per ogni istruzione, mentre in altre architetture, come CISC, non è vero.) Il disassemblatore lineare è però abbastanza *dummy*, non comprende il codice. Nel caso dei salti, non si preoccupa di dove saltare. Ad esempio: $v_1, v_2 | \ v_3, \ v_4 |$ se $v_1, v_2$ identificano un salto verso $v_4$ (l'istruzione macchina parte da $v_4$), ma il disassemblatore associa 2 *byte* = 1 *istruzione*, allora "salterà" verso $v_3$. Un esempio di disassemblatore lineare è `objdump -d`.
- **Interattivi**: Segue la semantica delle istruzioni. Se c'è *jump*, si riallinea a dove arriva il salto e continua con

l'analisi del codice macchina. Il set di istruzioni *escluse*,comprese tra *jump* e *dove arriva* viene lasciato all'utente. Questo perchè c'è un'*interazione* tra utente e disassemblatore, quindi se ottengo nuove conoscenze, le riverso nel disassemblatore e ottengo nuove informazioni. Se ad esempio riconosco una struttura dati, e la individuo, posso dire al disassemblatore di risolvere i riferimenti a questa struttura dati, in modo da non vedere più indirizzi generici, ma richiami a questa struttura. Un esempio di disassemblatore interattivo è **Ghidra**. Già nel fornire un file da analizzare a Ghidra, questi ci fornirà informazioni sul file (formato, architettura, compilatore etc....) che possiamo "dare per buono" o non seguire. Ci chiederà inoltre cosa usare per l'analisi, alcune "tecniche" sono assodate ed affidabili, altre, in rosso, sono sperimentali. Generalmente quelle proposte vanno bene per i nostri scopi.

## 32 vs 64 bit

Classifichiamo in *x_86* a *32 bit* e *x_86_64 / AMD64* a *64 bit*. La maggior differenza risiede nella dimensione dei registri (*4 byte* contro *8 byte*), ed alcune differenze tra registri o istruzioni.

## Intel vs AT&T

Nell'architettura *Intel* ho prima la destinazione (dove scrivo), poi la sorgente (dove leggo). Per *AT&T* è il contrario, abbiamo prima la sorgente (dove leggo), e dopo la destinazione (dove scrivo). Ricapitolando:

- Intel: `< destinazione,sorgente >`
- At&T: `< sorgente, destinazione>`

Esempio: `mov reg,8` quanti byte uso per rappresentare 8? dipende dal registro, se esso è a 32 o 64. `mov ptr,8`, quanto è grande l'indirizzo?

- In Intel l'istruzione è `mov DWORD ptr,8`
- In AT&T l'istruzione è `movl 8,ptr` dove con $l$ intendiamo *long*, 4 byte. Abbiamo anche 1 byte ($b$), 2 byte ($w$), e 8 byte ($q$).
- Con il suffisso `-M` vedo sempre i suffissi.

### Esadecimale, base 16

Si preferisce tale base perchè ha un mapping coi bit. Tra poco vedremo come. $257_{10} = 16^2 + 1 = (101)_{16} = 16^2 + 16^0$, è facile da convertirlo in binario, poichè basta scrivere ciascuno degli elementi sfruttando 4 bit : 0001|0000|0001. Questa scrittura viene compattata con l'uso di numeri da 0 a 9 e lettere da $a$ (1001) ad $f$ (1111).
Ad esempio: 1010|1011|0011|0111|1111 = $a|b|3|7|f$

## Big Endian vs Little Endian

Prendiamo $259_{10} = 256+3 = 103_{16} = |0001|0000|0011|$ Se raggruppiamo a blocchi di 8 bit avremmo: ...0001|00000011 = 1|3 (Quindi, se lavoro con 8 bit/2 byte, il numero più grande è `ff`, e ci posso rappresentare 256 byte diversi, ovvero $2^8$, da 0 a 255.) Con 8 bit dimezziamo lo spazio per memorizzare informazioni! Come rappresento 1|3 in **memoria**?

- Con **Big Endian** il byte più significativo va a sinistra, quindi in **memoria** scrivo 1|3.
- Con **Little Endian**, il byte più significativo va a destra, quindi in **memoria** scrivo 3|1.

In realtà, qui abbiamo supposto un *ordinamento totale*, in realtà potrebbe anche essere *parziale*, dato $v_1, v_2, v_3, v_4$ possiamo avere:

- Little Endian Intel totale: $v_4, ..., v_1$
- Little Endian parziale: $v_2, v_1 | v_4, v_3$
- Big Endian: $v_1, ..., v_4$

## Registri

Rappresentano memoria immediata che usa il processore.

In **X__86** abbiamo:

- *segmenti*: *CS* (dove sta il codice), *DS-ES-FS-GS* (data), *SS* (stack)

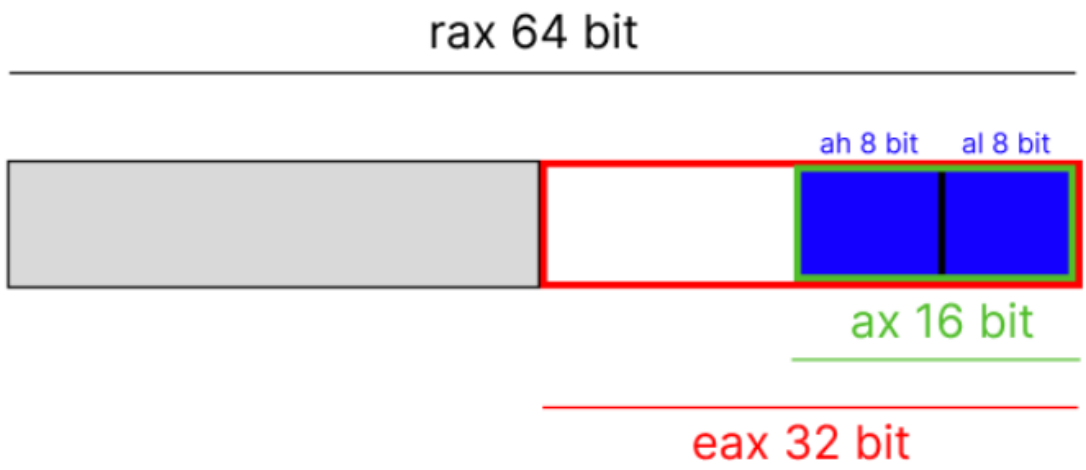
- *general purpose*: *eax* (accumulo) e sottoinsiemi *ax/ah/al ebx* e sottoinsiemi *bx/bh/bl ecx; edx; esi/si; edi/di; ebp/bp*
- *special purpose*: eflags (bit stato nel processore, interruzioni)/flags eip (instruction pointer) cr0, cr1...
- *floating point*: mmx0,...,mmx7 di tipo vettoriale usabili anche in virgola mobile, chiamandoli fpr0,...,frp7 xmm0,xmm15 anch'essi vettoriali.

La limitazione principale è che 32 registri sono pochi, quindi spesso si lavora con lo stack.

In **X_86_64** principalmente estendiamo la maggior parte dei registri:

- *general purpose*: *rax/eax/ax/ah/al*; *rbx; rcx*; *rsi*, ..., *rdi*, ..., *rbp*
- Aggiunta di nuovi registri *r8,...,r15*
- *special purpose*: *rflags*, *rip*.

Si lavora in pagine o segmenti. Non vedo i registri dei segmenti nel codice macchina, spesso sono *impliciti*. Ad esempio: `mov reg, dword ptr p`, in questo caso è implicito che, trattandosi di dati, uso il segmento *DS*. Se uso `call f`, con *f* funzione, allora uso *CS*, poichè si tratta di testo. Con `mov reg FS:[]` sto specificando di usare *FS*, altrimenti è sottointeso *DS*.




**NB**: L'Endianess si vede in memoria, **NON** nei registri! Quindi qui non mi preoccupo di dove collocare le cose.

# 10 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 spiazzamento 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, dove l'indirizzo non è completo, in quanto abbiamo soltanto l'*offset*.
Il secondo cerchio è un **indirizzo virtuale o logico**. 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. L'indirizzo logico diventa **lineare** se il segmento parte da 0, come vedremo tra poco.

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

E' un numero a 32 bit (se il file è a 32 bit), lo si vede come 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. (Perchè l'indirizzo logico è composto, come abbiamo detto, da segmento e offset, ma se il segmento parte da 0, allora mi serve solo l'offset per avere il dato completo.) Come lo capisco? se vedo i 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. (Lo stack parte dal basso, dove gli indirizzi di memoria sono grandi, e si espande verso l'alto, dove gl indirizzi di memoria diventano via via più 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 quindi definire una linea di divisione a priori.

---

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 assembler!
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.

# 2 novembre 2023

| | Basic Static Analysis | Advanced Static Analysis | Basic Dynamic Analysis | Advanced Dynamic Analysis |
|---|---|---|---|---|
| **White box** | *V* | *V* | | |
| **Grey box** | | | | *V* (più potente) |
| **Black box** | | | *V* | |

## Analisi statica

Quando parliamo di **Analisi statica**, abbiamo due categorie:

- **Advanced**: uso del disassembler interattivo, come Ghidra, in grado di analizzare il codice mediante interazione con utente.
- **Basic**: Si limita a guardare cosa contiene il file per rapportarsi al mondo circostante. Usa dei tool per capire cosa c'è dentro il programma, senza scendere al livello dell'assembler. Posso vedere API, system call, .dll usate, ma non istruzioni macchina. A volte basta questo. Cosa ci posso fare?
  - **hash**: firme digitali, usata in ambito forense, per dimostrare che un certo file o documento non è stato alterato.
    Possiamo calcolare con `sha256sum w03.exe`
  - Esistono siti che raccolgono *malware già noti*, quindi non ho bisogno di rifare il lavoro da 0. Tuttavia compaiono numerosi malware ogni giorno, quindi è difficile avere un database aggiornato. Gli *anti-virus*, per superare questo limite, identificano dei pattern particolari
    (*euristiche*) per l'identificazione. Il sito **Virustotal** contiene euristiche conosciute e ci dice se, un certo file, fornito da input, le contiene. Fornire un file a tale sito però, comporta aggiungere la firma del file nel sito, e quindi, chi ha creato il malware, può vedere se è stato analizzato su questo sito.
  - **stringhe**: cercare le stringe dentro un eseguibile, mediante comando `strings`, oppure `strings -n 2 nomefile` se cerco stringhe di due caratteri. Il malware può offuscare le stringhe!
  - **PE header**: dentro troviamo numerose informazioni, come le API. Ci sono vari software, in Windows `cffexplorer`, a cui passo un eseguibile con *drag&drop* e ne decodifica la testata. Oppure c'è `PEview`, `peBear`, `peTools`, `dependency worker` (prende un eseguibile e ricostruisce API che individua, con tutti i collegamenti, anche ricorsive. Non più supportato.). Infine `resourceHacker`, in cui possiamo vedere risorse incluse, come icone, manifest, ... tutte sostituibili!

Abbiamo questa fase quando il malware in esame è *fermo*, come fosse un cadavere.

## Analisi dinamica

Parlando di **analisi dinamica**, si ha:

- **Basic**: esegue monitoraggio, come WireShark. Viene visto anche ciò che viene scritto o letto dalla nostra applicazione. `peID` è un tools per analisi statica, ma usa plugin che potrebbero eseguire il codice, quindi non è proprio di analisi statica.
- **Advanced**: eseguita con Debugger, strumento più efficiente. Monitor che associa ciò che fa il programma con codice ad alto livello. Lenta e costosa, ma potente, il malware lo teme e cerca di proteggersi. Ne è un esempio **OllyDbg**.

Trattiamo questa fase ogni volta che viene richiesta l'esecuzione di un qualsiasi pezzo di malware, quindi non esaminiamo il cadavere come prima.

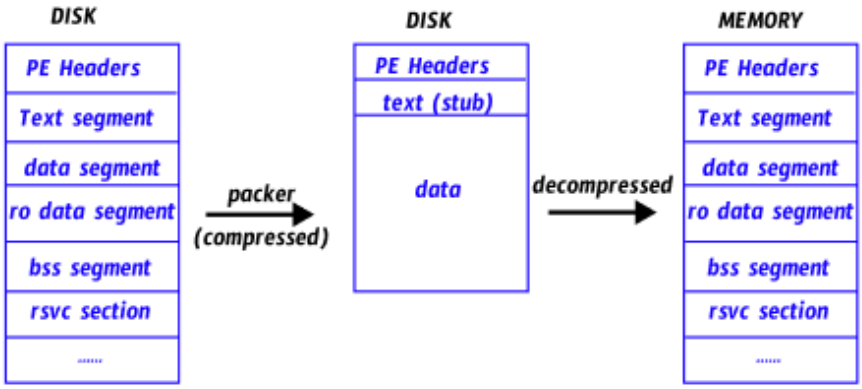
# 7 novembre 2023

La differenza tra analisi statica, rispetto all'avanzata, risiede nello sforzo per portare a termine l'analisi. La *statica* (qualsiasi tipo) è meno esosa (ma rende anche meno) rispetto ad una *dinamica*. Per l'*analisi dinamica di base* si richiede di eseguire il malware in un ambiente idoneo, ad esempio sistema operativo Windows. Non sulla macchina host, bensì un ambiente controllato, come un macchina virtuale guest.

## Process Explorer

**Tool dinamico.** Normalmente tutti questi tool di debug si eseguono come amministratore, perchè operano a basso livello. Parte facendo vedere tutti i processi del sistema, è dinamico, vediamo **chi lancia cosa**, quanto occupa. Quando lancio qualcosa, lo vedo qui insieme al *PID*. Molti malware cercano di nascondere i propri processi facendo finta di essere un processo del sistema. Esistono tanti processi di sistema, ne viene scelto uno di cui il malware è sicuro della sua presenza, allora il malware si fa passare per lui.
`services.exe` lancia i daemon, e si appoggia a `svc-host.exe`, che ospita il servizio. Tipicamente si impersonifica quest'ultimo. *Process Explorer* ci dice, con *tasto destro* → *properties* → *image file* → *verify*, la firma digitale rispetto al processo originale. Se è di Microsoft, allora è verificato. Abbiamo anche una scorciatoia per *VirusTotal*, ma se non siamo connessi dà errore di sicurezza. Altra cosa che fa è, da
*properties* → *strings* → *image*, guardare l'eseguibile e ritornare le stringhe.
Posso farlo sia sul programma sia sul processo in esecuzione (essendo *tool dinamico*). Normalmente le stringhe non differiscono di molto. Non coindicono quando l'eseguibile in memoria è diverso dall'eseguibile su file, ad esempio il malware lancia un eseguibile ufficiale (che è verificato), ma poi svuota tale eseguibile, e sostituendolo in memoria con il malware.

Sappiamo che un file eseguibile è composto da *testo, data, bss, risorse*.
Per risparmiare sulle dimensioni dell'eseguibile, si usano i **packer**, che prendono queste info e le trasformano in un nuovo eseguibile, con una parte *testo* piccolissima, e il resto è tutto compresso, decomprimendolo poi in memoria. Poi fa `jump` a prima cella del codice originale. Quando viene caricato in memoria, ricrea la forma originale. Se confronto con la versione compressa, ovviamente avrei un offuscamento, dobbiamo confrontare la versione non compressa nel disco (anche la versione compressa è nel disco) e memoria.




## Process Monitor

Eseguo anche lui come amministratore, ciò che fa è monitorare il sistema durante l'esecuzione, prende tutti i **processi** nel sistema. Possiamo applicare dei filtri (a forma di imbuto). Possiamo selezionare quali operazioni guardare. Interessanti sono i *registri di sistema*, con cui Windows raggruppa le informazioni di configurazione. Noi vediamo le **chiavi di registro**. Sono delle informazioni da preservare, e i *registri* ne contengono molte. Esiste tool di sistema **regit** che ci permette di modificare le chiavi (sono le *HKEY_LOCAL_MACHINE_xx*). Per vedere queste chiavi, c'è di meglio.

## Regshot

Esegue due **fotografie**, una a sistema pulito senza malware, e quindi *shoot* delle chiavi di registro. Quando ho finito, lancio il malware. Poi verrà eseguito il *compare* tra le chiavi, per vedere cosa è cambiato. Qualcosa cambia sempre, non è difficile capire ciò che tocca il malware rispetto ad altri processi. E' un tool lentino, perchè fa il check di molte chiavi. Quando si ha finito, otterremo un file (**da usare nel report**) con ciò che è cambiato.

### Wireshark

Usato per flussi a livello di rete.

### ApateDNS

Per il malware è meglio mettere nomi simboli piuttosto che indirizzi IP, con un server DNS che mappa il nome simbolico su indirizzo attivo. Quando seguo il malware, ogni qual volta che farà tale richiesta passerà per il DNS. Voglio poter decidere se, quando un malware fa richiesta DNS, io possa rispondere con un mio indirizzo, non voglio bloccare l'analisi. Spesso questo tool non si usa da solo, di solito lo si affianca a **Inetsim** (linux), il quale lancia una serie di server configurabili per rispondere alle richieste come voglio io. (ad esempio: se viene richiesto Google, ritorna address google. Se viene chiesto indirizzo *strano*, fornisco il mio indirizzo!)

### ApiMonitor

Ha più informazioni sulle API, ci aiuta ad interpretare ciò che fa l'applicazione. Si può usare come alternativa di ProcessMonitor. Entrambi condividono il limite di monitorare le applicazioni. Un deviceDriver usa API a basso livello, quindi solo ApiMonitor riuscirebbe a vedere qualcosa.

### Esempio di analisi w03-03 [parte 1]

Mai farlo sulla macchina primaria. In ogni caso dovrò fare snapshot. Lo snapshot in Qemu può essere:

- esterno: file che è derivato da quello principale, se lo tocco, tocco il derivato, non l'originale. Se tocco l'originale, quello derivato si corrompe. In Qemu c'è `qemu-img create -b <NomeSnapshot> <NomeOrig>`, quindi il file si appoggia ad un altro.
- interno: nel file registro i punti di recupero, come `qemu snapshot -c`

Normalmente si fanno due snapshot, dal primo si disabilitano alcune impostazioni di sicurezza, e poi si fa il secondo snapshot. Se andiamo in *sicurezza di Windows*, dove abbiamo le varie protezioni di virus, minacce, protezione account. Dobbiamo togliere *"controllo delle app per il browser"* (togliere *protezione del flusso di controllo CFG* e *protezione esecuzione DEP*, anche la randomizzazione può essere utile da rimuovere.) Dobbiamo togliere anche *protezione in tempo reale* (presente in *Impostazioni di Protezione da virus e minacce*). Anche *invio automatico dei file di esempio* etc sono cose tranquillamente rimovibili.

#### Come passo il malware su VM?

Il prof usa *shared_memory*, ma questa non è una buona pratica. Posso usare una pennetta usb. Anche le *guest additions* possono far capire al malware di trovarsi in una VM.

#### Inizio analisi

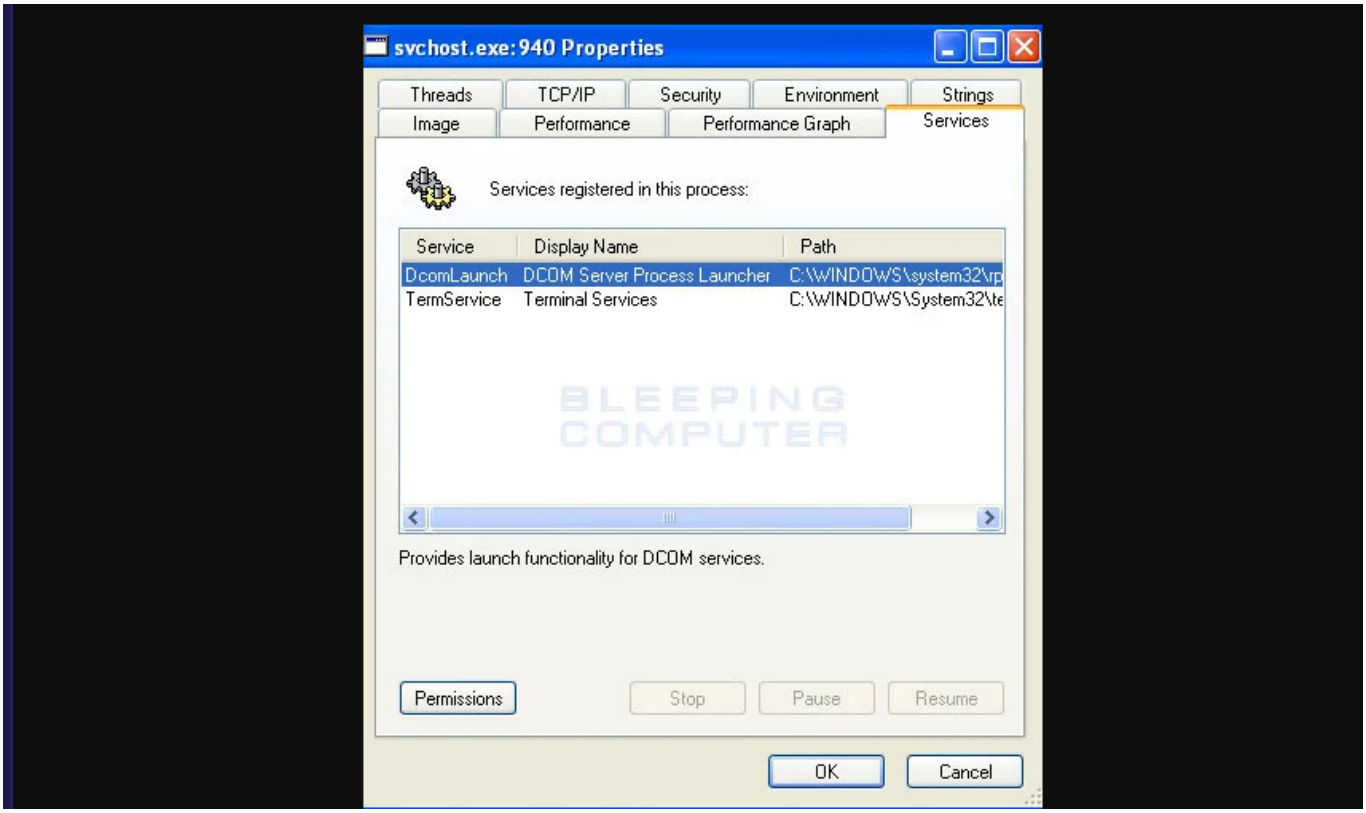
Avviamo *process monitor* e *process explorer* a 64bit, tanto c'è compatibilità. Lanciamo il malware. Compare un `vc-host.exe`, ho finestra errore che mi dice "impossibile avviare correttamente...". Abbiamo creato quindi questo processo, allora il malware vorrebbe fare la sostituzione. Se vedo le *properties* mi dice che è *verificato*, perchè si vede il file eseguibile da dove è partito. Allora questo malware ha livelli di accesso kernel. Se c'è questa sostituzione, ho anche stringhe diverse, infatti non ci sono! Perchè abbiamo cose diverse, e perchè il malware si richiude subito. Il malware non sta funzionando, perchè è un malware vecchiotto, lavora a basso livello, e c'è incompatibilità con Windows 10. Ne serve una più vecchia, ad esempio Windows XP. Dovrei partire sempre da Windows vecchi? No, avrei tools vecchi.

## 9 novembre 2023

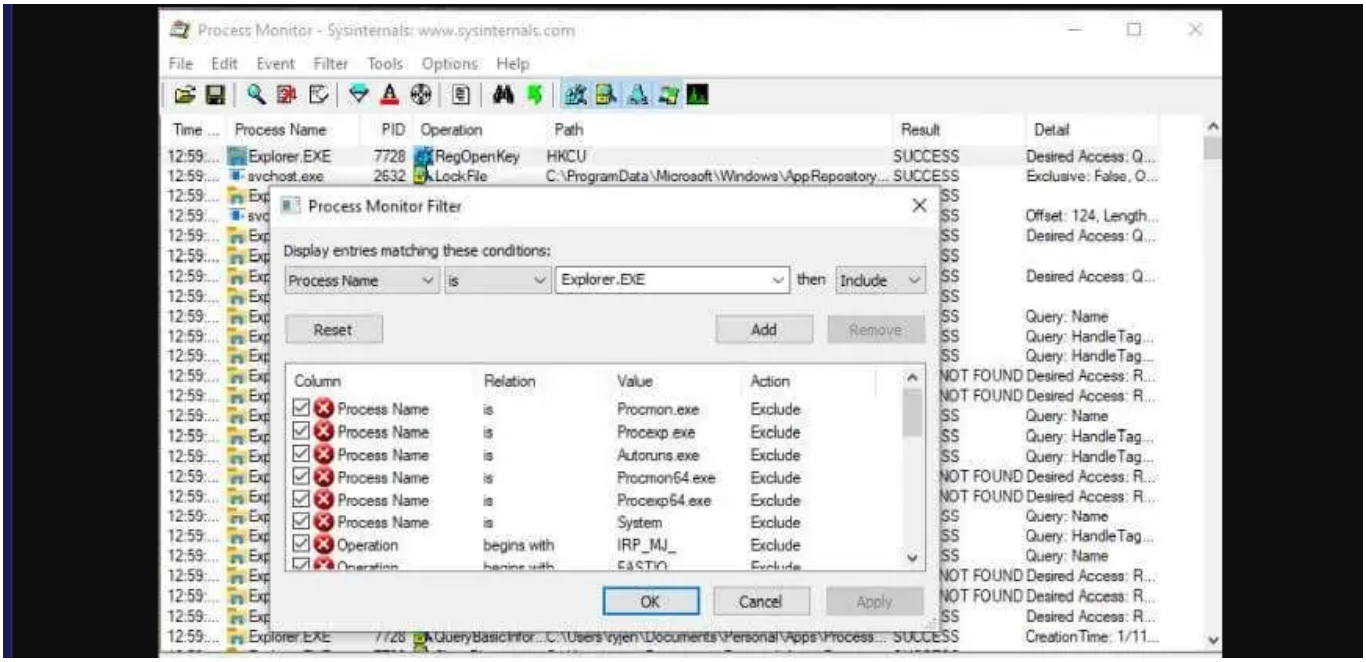
### Esempio di analisi w03-03 [parte 2]

Proviamo Windows XP su VirtualBox. Quasi nulla la resistenza al malware. Si fa sempre snapshot prima di scaricare il malware. Per procedere, eseguiamo lo stesso iter già visto con Win10: *Process Explorer* (lanciato come amministratore) e *Process Monitor* (sempre come amministratore).

Lanciamo *explorer.exe*, abbiamo processo `vc-host.exe` visibile su *Process Explorer*. Cosa vogliamo da questo malware? Vorremmo capire cosa fa, a che cosa serve, senza fare un'analisi dettagliata del suo contenuto. Il malware è identificato da un numero, 724, creato da *lab03-03*. Nel *filter*, possiamo dire che siamo interessati agli eventi generati da 724 (nella foto sotto, presa online, sono 940), escludendo gli altri. Possiamo vedere se tale processo è conosciuto da *VirusTotal*, se non ho internet calcolo lo sha e lo metto sul sito. Alternativa: vedo le *properties*. Qui risulta *verificato*, perchè `lab03-03` ha creato il processo `vc-host.exe` che è ufficiale, lo ha svuotato e ci ha messo ciò che voleva lui. Se apriamo un `vc-host.exe` ufficiale, vediamo le stringhe tra *immagine* e *memoria*, esse sono assolutamente uguali. Sulla versione `vc-host.exe` con malware, c'è invece differenza.




Troviamo alcune stringhe "curiose": *shift*, *backspace*,... si tratta di un **Keylogger**. Ovvero salva le operazioni eseguite dall'utente. Però non basta, dobbiamo dimostrare questa tesi. Se prende ciò che l'utente scrive, allora lo salva da qualche parte. Qui entra in gioco **process Monitor**, che registra le API usate dal processo.




Possiamo vedere se queste informazioni vengono mandate tramite *internet*, ma non vengono scritte lì. Dobbiamo prima

includere il nuovo filtro con `add` e poi fare `apply`. Alternative: *file su disco*, *registri del sistema* (che sono sempre file di sistema, ma organizzati dal sistema operativo). Ci sono degli eventi, ma se vediamo le chiavi, notiamo che molte non le apre oppure sono associate al terminale. Diciamo che sono operazioni *comuni* a tutte le applicazioni. Vediamo sui *file su disco*, vediamo un accesso continuo ad un unico file di log sul desktop. Infatti, sul desktop è comparso un nuovo *file.log* con tutto ciò che è stato scritto.

## Analisi ed intro al Debugger

- **White-box**: Composta da analisi *statica di base* e *avanzata*.
- **Black box**: *dinamica di base*.
- **Grey box**: eseguo programma in maniera controllata, si usano i *debugger*.

Tipicamente i debugger lavorano a livello del codice sorgente. Classifichiamo i debugger in due famiglie: **Source level** e **Assembly level**.

Altra differenziazione è su *come operano*: **User Mode** vs **Kernel Mode**.

Ovviamente il secondo è molto più potente, perchè può debuggare anche cose di tipo kernel. Poichè il malware è scritto per l'utente, spesso basta *user mode*. I *rootkit* (processi che non vediamo mai nei programmi visti, perchè lavorano a livello di sistema operativo, nascondendosi), lavorano in *kernel mode*, quindi serve debugger di tipo kernel.

Alcuni debugger lavorano in modalità *locale*, altri in *remota*, o ancora *mista*. Il locale gira sulla stessa macchina del processo esaminato, nella modalità remota c'è disaccoppiamento. Esempio in questo secondo caso, malware *Android*, il debugger lo metto su un computer, è sicuramente più comodo.

Ultima distinzione: malware con *GUI* (interfaccia grafica) ed altri basati su *CLI*, cioè linea comando. I debugger hanno comunque "più potenza" a livello CLI, poichè essa è la base, e sopra si costruisce la GUI. Questa potenza in più è data dal fatto che con GUI è difficile riportare tutte le complicazioni graficamente.

Il primo debugger che vediamo è quello incluso in **Ghidra**. In realtà è un *meta-debugger*, cioè capace di integrare nel proprio flusso di lavoro i risultati di un debugger separato. Quando importiamo un programma, clicchiamo sul *bacarozzo* vicino al *drago*. Se non c'è, lo abilitiamo nel campo *tools*. Ciò che ci si pone davanti è la *finestra del disassembler*, e tutto il resto è *debugger*. In *debug target* (in alto a sinistra), specifico quale debug usare. I debugger proposti sono:

- *gdb* (tre versioni: locale, via ssh, via gadp)
- *ldb* (debugger per MACOS, anche qui meccanismi locali e remoto)
- *windbg* (debug di Windows, sia 32 sia 64 bit, sia user sia kernel, sia local che remoto... insomma tutte le versioni)
- *windbg Preview* o *windbg 2* (versione successiva)

Tuttavia lavorare con queste implementazioni in Ghidra non è semplice, si registra una *traccia dell'esecuzione* su cui scorrere avanti ed indietro.

Anche **IDA Pro** è valutabile, perchè include un proprio debugger, oltre a funzionare con *meta-debugger*. Sia *user* sia *kernel*, sia locale sia remoto, sia 32 sia 64 bit.

Un altro debugger è **SoftIce** puramente *kernel mode*, è tipo un *hypervisor* tra *kernel* e *sistema operativo*. Potentissimo quanto vecchio. Oggi inutile.

Oggi si preferisce usare **OllyDbg versione 1.10**, anche lui vecchio, ma comunque con molti plugin (esempio: auto identificazione di strutture dati note). E' solo *usermode 32 bit Intel*, di tipo *GUI*. Da lui deriva *OllyDbg 2*, interfaccia uguale, ma riscritto internamente. Lento, pochi plugin. Il creatore poteva evitarselo. C'è anche *OllyDbg 64*, però meh pure lui. Abbiamo anche *ImmuneDbg*, c'è versione di base oltre a quella a pagamento. Stessa interfaccia, è riprogrammabile in Python, cioè posso scrivere script in Python per automatizzare l'analisi. Altro figlio è *X64dbg*, sia a 32 sia a 64 bit. Interfaccia diversa? Ovviamente no. E' buono per i 64 bit.

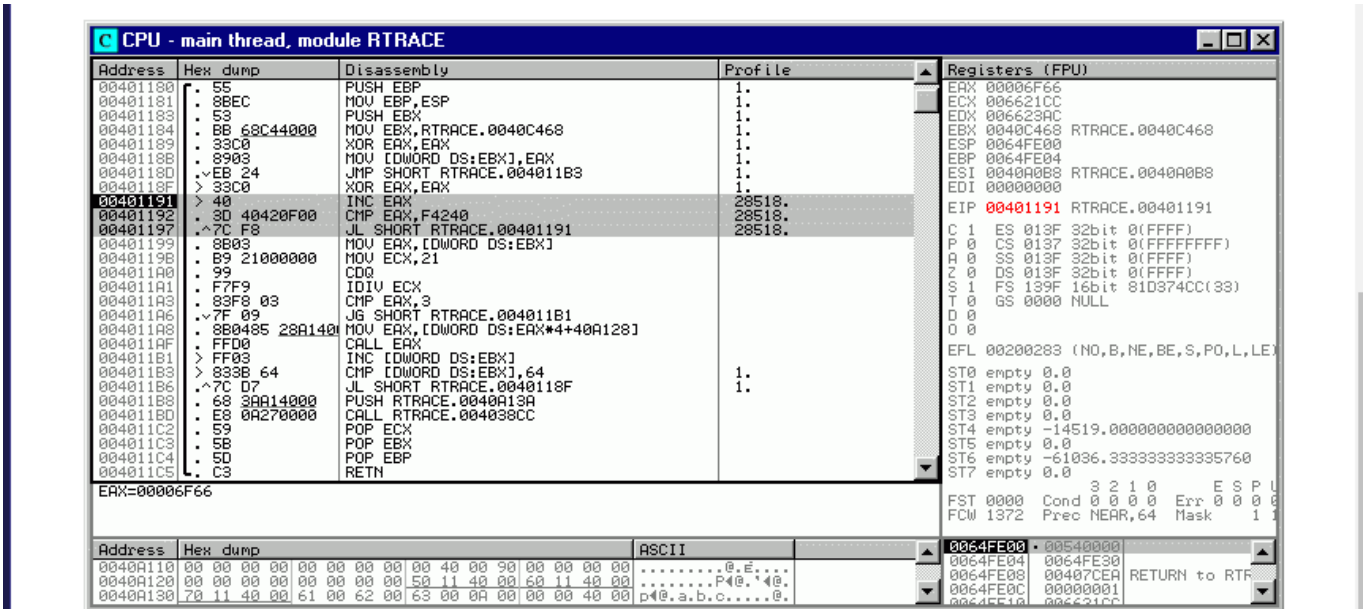
L'idea è che, a seconda di ciò che abbiamo avanti, dobbiamo ponderare la scelta del debugger migliore.

## Overview su OllyDBG

- *In alto a sx*: disassemblato, parte da entry point, sintassi intel (`<destinazione,origine>`).
- *In alto a dx*: finestra registri, il loro contenuto, tutti! (Esiste il comando `follow in dump` per seguire uno specifico registro). Interessante è *debug flag T*, bit che, quando viene generata eccezione, viene impostato ad

1 prima di eseguire, genera trap per il sistema operativo , che invece di segnalarla al processo (che sarebbe il naturale corso degli eventi) la segnala al debugger. Con il comando *step into F7*, decidiamo noi come proseguire in questi casi.

- *In basso a sx*: finestra sui dati, vedo quello a cui punta un registro, vedendo la sua evoluzione.
- *In basso a dx*: Finestra sullo stack, che cresce per indirizzi decrescenti.




OllyDbg interpreta anche i valori. In alto a sx c'è il tasto *play* per eseguire l'eseguibile sotto al debugger, posso sospenderlo ovviamente. Ad esempio, con *W03* possiamo bloccarci nell'esatto momento in cui si sta creando la finestra. Possiamo fare anche *step into - F7*, ovvero istruzione per volta, aggiornando i registri (interessante è il flag *T* visto sopra). Se facciamo una call, con *step into*, vedo istruzione per istruzione, con *step over* tale call viene vista come un unico blocco di codice, passando alla call successiva ad esempia. Il malware potrebbe "capire" che stiamo usando *step into* e bloccarlo, allora usiamo *step over* con *breakpoint*, per aggirare il problema. Nell'immagine in memoria, il debugger, prima di lanciare istruzione, sostituisce un byte con singolo byte, in particolare usa *int3*, che genera istruzione di *trap*. Poi il debugger rimette in *int3* l'istruzione originale, se decidiamo di eseguirla. Possiamo fare patch (tipo NOP 90), ma le starei mettendo in memoria, modificando il processo in memoria, non l'eseguibile, quindi modifico il comportamento "in quel momento".

## 14 novembre 2023

Continuiamo con *OllyDebug*, prendiamo *W03*. Concetto essenziale è quello dei *breakpoint*. Permette al debugger di intervenire, quando ritiene opportuno. Sono i meccanismi principali usati dal malware per capire se sta venendo analizzato. Classifichiamo:

- Breakpoint attivati in esecuzione: i più comuni.
- Breakpoint attivati quando accedo ad una locazione in memoria.
- Breakpoint software, tramite sistema operativo.
- Breakpoint hardware.

I **breakpoint Execution/Software**, basati su istruzione `INT3`, o `0xcc`. Un programma è composto da tante istruzioni macchina. Ad un certo punto divido per 0, quindi questa istruzione macchina, a livello hardware, genera un'interruzione per trasferire il controllo ad una routine dentro il sistema operativo. Il sistema può quindi gestirla in diversi modi, dal terminare il processo all'associazione di un gestore (presente nel codice stesso, fatto da noi). Riguarda processo/programma, sistema operativo, hardware.

Adesso, quando si verifica tale evento, il sistema operativo ferma l'esecuzione e demanda la gestione/controllo al debugger. Il debugger può fare quello che preferisce. Abbiamo due tipi di gestione:

- *first chance* : Vengono aggiunte `debug exception INT3`.
- *last chance*: Abbiamo interruzioni come divisione per 0.

Una tecnica di difesa del malware consiste nella ricerca in memoria del byte `0xcc`. Quando lo trova, e "capisce di non avercelo messo lui", termina, oppure cambia l'esecuzione originale.

Nei breakpoint **Access/Software**, il programma tenta ad accedere a delle pagine, che tramite debugger e sistema operativo, sono state poste ad esempio in sola lettura. Quindi si verifica un page fault. Ciò genera eccezione, la quale viene gestita, e successivamente vengono riassegnati i diritti originali su quella pagina.

Queste operazioni non hanno nulla a che fare con **Execution Step by step F7**, che sfrutta una *trap flag*, in cui dopo ogni istruzione macchina interviene il debugger, e l'esecuzione si ferma. Il malware può sempre vedere l'impostazione di questo trap flag, e poi vede se lo ha messo lui o meno.

Con `F7` usiamo trap flag, e andiamo di istruzione in istruzione, mentre con i `breakpoint` tipicamente eseguiamo il programma fino ad un certo punto (quindi eseguendo più istruzioni insieme).

I breakpoint **hardware** prevedono che il processore abbia registri contenenti un indirizzo, una dimensione, ed una modalità di accesso. Quando arriviamo ad un certo indirizzo, e proviamo a scrivere un byte ad un indirizzo (dimensione), genera una eccezione. In Hardware, verrà generata una eccezione hardware. Sono utili perchè il malware si difende maggiormente dai breakpoint software rispetto a quelli hardware. Non funzionano nello stesso modo. Di contro, il numero di registri hardware di debug è limitato. Non è detto che il malware non li controlli!

## Visione codice Ollydebug

Si parte da entry point, possiamo procedere con *step into*(F7, singola linea) e *step over* (F8, singola funzione/blocco di codice). Con *step over* usiamo in modo congiunto i breakpoint. Quando supera l'istruzione, rimuove il breakpoint. Con F2 mettiamo un breakpoint "stabile" di esecuzione. Con tasto destro → *breakpoint* → possiamo mettere un breakpoint hardware. Inoltre è possibile vedere la lista di hardware breakpoint. Abbiamo anche breakpoint **condizionali**, ad esempio in un ciclo, possiamo essere interessati solo ad una determinata condizione. Quindi le iterazioni non di nostro interesse sono sempre presenti, ma non ci vengono fornite; solo quella da noi specifica ci restituirà il controllo. Per i *breakpoint in memoria* dobbiamo accedere al sottomenu *dati* (nell'interfaccia, in basso) mentre i *breakpoint in esecuzione* li metto nella zona superiore dell'interfaccia. E' possibile vedere anche:

- L: log di tutto quello che è stato fatto, fin dall'apertura.
- E: tutti i moduli eseguibili, dimensione entry point,...
- M: memory map, come divisione in segmenti
- T: programmi Windows composti da più thread, qui li vediamo, li blocchiamo etc..! Quando debug ferma processo, li ferma tutti, a meno di non specificare un certo thread da bloccare.
- W: Tutte le finestre, come TextEdit, buttons,..., tutti gli elementi della finestra.
- H: tutti gli handle, possiamo vedere i token aperti dal programma.
- I: è possibile modificare l'eseguibile, in tale finestra vediamo le patch applicate.
- K: insieme di funzioni invocate come risultano sullo stack.
- B: breakpoint, posso assegnare label agli indirizzi, anche se spesso si preferisce riportarle sul disassembler.
- S: mano a mano che eseguo il codice, mi fa vedere sorgente e dove mi trovo. Non lo useremo, in quanto non si ha il sorgente.

Da *debug* possiamo "animare il codice", quindi vengono eseguite le istruzioni, vedendo il risultato. Poco utile. Più utile sono le *tracce*, utili per registrare un'esecuzione, per poi rivederle con calma. Quindi usiamo una tecnica basata su *temporizzazione*.

Se siamo dentro una funzione, possiamo fare *esegui fino a return, F9*, quindi trova il return della funzione, mette il breakpoint, e si ferma. Se siamo dentro *.dll*, possiamo fare *return fino a user code*, quindi ritorna appena esce.

NB: **F4 significa *esegui fino al cursore***

## Come trovare winMain

L'esecuzione con F8, se non ritorna, indica che il *winMain* è interno alla funzione. Allora uso F4 fino ad una certa funzione, e poi F7 per entrarci dentro. Voglio trovare invocazione con F8, perchè passo sopra alla call, anche quelle sconosciute. Finchè il programma non si ferma, non è winMain. Quando vedo che il *programma parte* (quindi vedo la finestra), ho trovato qualcosa di interessante. Me lo salvo con F2, e vado avanti con F8, per cercare ciclo messaggi.

Procedo così finchè non trovo un ciclo, lo vedo perchè ripercorre ricorsivamente delle istruzioni. Mi metto col puntatore sulla jump, e faccio F4, per saltare direttamente senza aspettare. Poichè andando avanti riparte il programma, uso F2 per salvarlo. Entro nelle funzioni con F7, se eseguo con F8, procedo, ad un certo punto trovo il ciclo dei messaggi (possiamo anche *animarlo*).

### Lab 09-01 [parte 1]

Vogliamo capire come è fatto tale malware. Debugger è analisi avanzata, partiamo quindi con analisi di base statica. Ottimale è *PE-Studio*, nell'esempio usiamo *PE-Explorer*. Vediamo info di base, come *entry point*, data creazione, etc. Troviamo le sezioni che compongono il programma, come *read only* e *data*. Con *PE-bear* (si legge meglio), vediamo che il programma non è offuscato, perchè le dimensioni delle sezioni coincidono (vedo *virtual size* e *raw*). Se differenza importante, allora offuscato. Niente info utili, passiamo ad *analisi dinamica di base*. Facciamo snapshot, e poi apriamo *Process Explorer*. Apriamo *Process Monitor*. E richiudiamo.

Andiamo con *Ghidra*. Proviamo a cercare le stringe. Potrebbero essere interessanti *NOTHING, HTTP, DOWNLOAD, UPLOAD, SLEEP, CMD, un sito Http*, che ci suggeriscono delle attività fatte su un server esterno. C'è una stringa che suggerisce la stampa di qualcosa. C'è una chiave di Registro *XPS*. Per avere persistenza, il malware usa chiavi di registro. Non può usarla direttamente, perchè viene sovrascritta tra comandi leciti. Può usarla in modo fittizio. Ad esempio, *SOFTWARE MICROSOFT XPS* invece di *SOFTWARE MICROSOFT XPS*. (La differenza è nel doppio spazio nel secondo nome.) E' presente anche `$SYSTEMROOT\$`, variabile d'ambiente dove vive l'installazione del sistema operativo. In particolare, si fa riferimento a `\system32\`, che insieme a *cmd.exe* potrebbe sembrare un accesso a shell comandi. I comandi sono di tipo *dos*, tra cui c'è `/c del`, per cancellare un file. Se facessimo da prompt `cmd.exe /c date` mi ritorna la data da una sottoshell. Con `/c del` cancelliamo un file. Tutto questo perchè le stringhe non sono offuscate.

Ora possiamo tornare ad analisi dinamica di base, con *Process Explorer* e *Process Monitor*. Dovremmo anche lanciare *Regshot*, per fare confronto tra snapshot pre-malware e post-malware.

Eseguiamo il malware, che si è aperto e poi si è cancellato. E' ancora in esecuzione? Vedendo su Process Explorer, no. Probabilmente si è cancellato perchè qualche cosa non gli è piaciuta.

Sappiamo il nome del processo, contiene "lab-09". Quindi da *Process Monitor* filtriamo e vediamo cosa ha fatto. Quali sono gli eventi legati all'esecuzione? Sappiamo, dall'analisi statica, che si interagisce con la chiave di sistema. Applichiamo il filtro per vedere solo chiavi di registro, cercando "microsoft" (con lo spazio). E' presente, la apre ma non la trova (NAME NOT FOUND), quindi non l'ha creata. Vediamo inoltre che la chiave è sotto la cartella "WOW6432" perchè malware 32 bit, sistema a 64 bit, e c'è meccanismo di alias. Non viene ulteriormente usata. Sappiamo che si è cancellato, con shell di comandi. Vediamo di fare una ricerca sui processi, come quelli che governano i file.exe.
Troviamo un *process create*. Con tasto destro e *property* vediamo linea comandi: confermiamo che il malware cancella se stesso. Quindi al malware non piaceva l'ambiente, e si è cancellato. Non possiamo andare avanti, serve il debugger. Prossimo step: *analisi dinamica avanzata*.

## 16 Novembre 2023

### Lab 09-01 [parte 2]

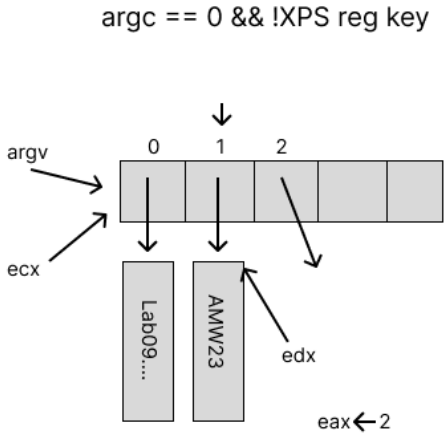
Se vediamo, in Ghidra, l'entry-point del decompilato, sappiamo che, verso la fine, dovremmo trovare la funzione main. Come la individuo? Prende due parametri, ovvero `argc` e `argv`, trattandosi di un programma da "terminale", senza GUI. Trovato il main, a riga `00402af0`, facciamo partire **OllyDgb** e saltiamo a tale istruzione, mediante `CTRL+G`. Mettiamo breakpoint con F2, e facciamo partire l'esecuzione, passo passo. Quando trovo una call, ci passo sopra, non entro dentro (questo lo farebbe il debugger), fino a che non trovo qualche cosa di particolare, come *processo terminato*. Il debugger ha tenuto aperto il sistema operativo, quindi non si è autodistrutto. Possiamo fare restart e ricominciare l'esecuzione, run fino all'entry point e poi f8. Stavolta mi fermo alla call che ha portato alla chiusura, a riga `402410`. Vediamo su Ghidra cosa fa questa funzione, invoca `GetModuleFileName`, che ritorna il percorso dalla root in poi, del modulo specificato. Quindi il percorso di Lab 09-01.

Poi abbiamo `getShortPathName`, ovvero il nome del file finale. Successivamente `ShellExecute`, comando mediante shell dei comandi. Il comando è `cmd_exe`, che sta trai parametri. Vediamo anche le altre stringe `\c del`, `>> null` (per ridirezionare l'outuput in un "pozzo oscuro"), quindi sta ricreando la stringa per cancellare il file. Alla fine invoca una serie di funzioni che ci portano alla fine a `FUN_00402e3c`, che richiama `ExitProcess`, la quale uccide il processo. Quindi la funzione da cui parte tutto, dopo riga`40240e`, è quella che effetta il suicidio. E' buona norma mettere etichette anche nel debugger. In **OllyDbg**, a partire dal main (`00402af0`), vediamo un salto a `00402801` che ci evita

di passare per il suicidio. Torniamo su Ghidra per capire come entrarci. In Ghidra rinominiamo la funzione main, cioè `int main(int argc, char ** argv)`. Vediamo che viene confrontato `argc` con 1, ovvero se `argc=1` vuol dire che non abbiamo argomenti, in quanto il primo "argomento" sarebbe il nome del file, quindi almeno un argomento ci sarà. A riga `00401023` c'è un altro confronto, ci interessa perchè è un altro confronto per entrare nel suicidio. In questa funzione, in Ghidra, vediamo un riferimento a `0040c03e`, ovvero la chiave *farlocca*. Dinamicamente, con F8 su OllyDbg, eseguendo troviamo `EAX=0`, quindi faremo suicidio. **Il programma si uccide perchè numero di argomenti è zero, e non trova una chiave che si aspetta di trovare.**

Lo abbiamo scoperto mediante analisi debug, analisi statica e dinamica. Se ne rispettassi solo una, funzionerebbe? Potrei mettere la chiave, ma probabilmente avrei altri check sul contenuto. Proviamo a mettere un argomento. Altra strategia sarebbe modificare i test, modificando i valori. Ad esempio, a riga `00402B08`, il salto viene preso perchè flag `Z=1` (lo vediamo nel campo Registers, a destra del programma.) Possiamo, nel campo registri, dove si trova Z, premere tasto destro e fare `reset`. Potrei anche modificare `EAX` e metterci il valore `1` invece che `0`. Posso modificare registri di stato come preferisco. Tornando alla tecnica originale, vado su `debug -> arguments` e passo argomento per eseguire. Passiamo un valore a caso, perchè non so cosa voglia. Riusciamo a superare il primo jump. Ad altezza media dell'interfaccia, vediamo che lo stack ci informa delle operazioni svolte. Andando avanti, cadiamo comunque nel suicidio, quindi non basta mettere un argomento, ma vuole qualcosa di specifico. A `00402510` avviene ciò, su Ghidra, vedendo il decompilato, sicuramente più comprensibile dell'assembly. Se vediamo da dove siamo partiti, da riga `00402b1d`, mettiamo su `EAX` ed `ECX` *argc* ed *argv*, in `EDX` l'indirizzo della stringa che corrisponde all'ultimo argomento.

- *argv* punta a vettore di stringhe, quindi ogni elemento è a sua volta un puntatore ad una stringa.
- In `EAX` abbiamo messo 2 (perchè abbiamo passato un argomento).
- In `ECX` viene messo l'indirizzo del vettore, cioè `argv`.
- In `EDX` faccio degli spiazzamenti, in particolare `ECX+4`, cioè un indirizzo. In `EDX` quindi puntiamo all'*indirizzo della stringa*, con `LEA` avremmo puntato all'*indirizzo contenente il puntatore della stringa*. (risata malvagia).




Nel decompilato di `FUN_00402510`, rinominata `check_password`, vediamo che vengono eseguiti continui check, che impostano `uVar2` a 0 se qualcosa non gli piace. Però, senza perdere tempo, vediamo che alla fine a noi va bene se `uVar2 = 1`, inoltre capiamo, dai check che fa, che la password che vuole è `abcd`.

La cosa migliore da fare è *una patch*, possiamo farla con **OllyDbg**. Devo assemblare delle nuove istruzioni, se premiamo "spazio" su una istruzione, possiamo metterne una nuova. Mettiamo `XOR EAX, EAX`, il quale lo azzera, poichè lo XOR ritorna 1 se e solo se gli ingressi sono diversi tra di loro. Quindi, essendo tutti ingressi uguali (confrontiamo `EAX` con sè stesso) ritornerà sempre falso (0), azzerandolo.

Ricordiamo che vogliamo avere `1` su EAX alla fine, quindi, dopo l'azzeramento, come seconda istruzione `402512`, mettiamo `INC EAX`, per ottenere 1.
La funzione è di tipo `___cdecl`, quindi gli argomenti vengono tolti dal chiamante. Nella riga successiva `402513`, mettiamo `RETN` (lo fa lui in automatico). **Tali modifiche avvengono sull'eseguibile in memoria, non sul disco.**

Per **salvarlo**, tasto destro sulla parte di codice, `copy to executable`, e poi `all-modification`, voglio copiare l'intero eseguibile. Infine `copy all`. Si apre una seconda finestra, con il codice. Questa va salvata, tasto destro, `save file`, e quindi la salviamo chiamandolo in modo riconoscibile. Chiudiamo il file originale, e mettiamo il file patchato. **Non mantiene i breakpoint**. Devo rimetterli tutti, per questo è buona norma salvare le cose anche esternamente, come Ghidra.

Facciamo ripartire il programma, mettiamo breakpoint sul main, passiamo argomento `AMW2023`. Finalmente abbiamo superato il suicidio. Ora? Magari, con analisi dinamica di base, posso ottenere altre informazioni. Riapriamo **Process Explorer, Process Monitor64, Regshot**. In quest'ultimo prendiamo la fotografia pre-lancio, poi lanciamo il programma, e prendiamo fotografia successiva.

**NB: meglio lanciare una copia dell'eseguibile patchato.** Si è richiuso e cancellato nuovamente! **Non abbiamo patchato il controllo del parametro.** Da Shell, lanciamo il programma patchato con il parametro di prima (non lo abbiamo patchato). Il programma si è richiuso, vediamo se qualche tool ha preso qualche dato utile. Con **Process Monitor** filtriamo gli eventi per `file system`. Spesso quelli in fondo sono quelli di nostro interesse. Niente di che.

Proviamo con **Regshot**, ci dice 21 valori modificati. Tuttavia sembrano modifiche del sistema operativo, la *registry key* non sembra essere stata toccata. Qui spesso si va di esperienza: molti malware richiedono *diritti di amministratore*. Noi lo abbiamo lanciato come *user*, magari questo malware funziona bene con *admin*. Potrei capirlo se, vedendo i *log*, trovassi qualche failure dato dai mancati diritti. Proviamo con *admin*.

**NB: Sicurezza di Windows -> Impostazioni di protezione da virus e minacce -> Disattiva protezione in tempo reale**. Non è persistente. Lanciamo come *root* `cmd.exe`, poi ci spiazziamo nella cartella contenente il malware. Prima rifacciamo gli snapshot, perchè stiamo provando un nuovo avvio con settings diversi. Da `cmd.exe` lo lanceremo normalmente (abbiamo già eseguito come amministratore `cmd.exe`). E' stato nuovamente rimosso. Facciamo il secondo shot. Nuovamente, nessuna conseguenza. Cosa sbagliamo? Magari, lui guarda il numero di argomenti, se sono 0 chiudi, se 1, metti una password. Ma magari si aspetta altri argomenti. Sennò perchè specificare di prendere l'*ultimo argomento*?

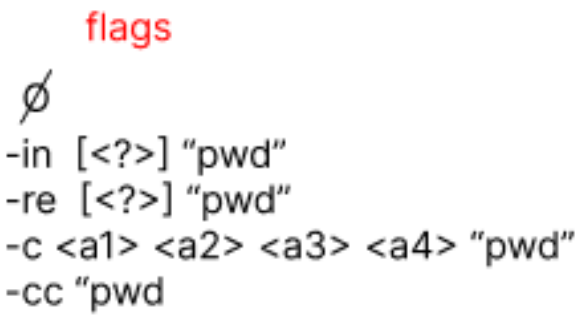
Torniamo al *disassembler*, nel *main*. Vediamo che il suicidio viene fatto 7 volte, noi ne abbiamo trovati solo 2. Quindi probabilmente siamo entrati in uno degli altri 5. Possiamo vedere che `argv` viene chiamato svariate volte localmente. Ad esempio in `00402b3f` accediamo al primo argomento. Poi viene fatto push ad un indirizzo, che è un dato `DAT_0040C170`, che è `-in`, vediamo sopra che ce ne sono altri. Allora `data -> terminateString` per creare tali stringhe (Ghidra non le riconosce perchè troppo brevi). Scendendo vediamo che Ghidra riconosce `strcmp`, cioè wrapper che richiama `strcmp`. Allora la funzione che richiama tale funzione la rinominiamo proprio così. Se le stringhe diverse, faccio confronti vari, e poi fa suicidio. Il nostro parametro (AMW2023) non va bene in questi check. In OllyDbg non viene riconosciuto *strcmp*, è molto più complicato usare solo lui! Ghidra, disassembler interattivo, mi da un aiuto enorme. Lo capisce perchè è codice, Ghidra è configurato in maniera tale da avere vari motori di analisi, tra cui `functionID`, che riconosce funzioni degli eseguibili confrontando con funzioni di libreria avente una certa firma. Non è una cosa banale, tipicamente può capitare che il `codice.o` venga rilocato, oppure che richieda altri riferimenti. Quindi non c'è una corrispondenza 1-1, ma cerca parti del codice *non modificabili*, e cerca solo loro. Ovviamente, se alcune cose non sono del database di Ghidra, non le trova. In `file->file configure->core->FidPlugin`, attivo di default. Allora ho `tools->functionId`, che mi dice le librerie che conosce. Altre cose, non riconoscerebbe le hash. Posso aumentare queste conoscenze. Se installo `mingw` di Windows, posso ricostruire database anche per quello.

# 21 Novembre 2023

## Lab 09-01 [parte 3]

Abbiamo visto che il malware si suicida. Ciò è dovuto dalla mancanza di alcune condizioni, infatti abbiamo visto che il malware si aspetta di essere lanciato con degli argomenti specifici. L'ultimo argomento è una password. Se corretta (noi l'abbiamo patchata), procediamo, e vediamo il primo argomento `argv[1]`:

- se è pari a `- in` (ovvero `install`: mediante `strcmp`, se 0 sono uguali), vede se il numero degli argomenti è 3 (nomefile, "*-in*", password). Vediamo che accetta anche un quarto parametro opzionale, tra "*-in*" e *password*.
- se è pari a `-re`, (ovvero `remove`), sempre 4 argomenti.
- se è pari a `-c`, (ovvero `configure`) può avere 7 argomenti (quindi 4 argomenti, togliendo la stringa, il nome del file, e la password.)
- se è pari a `-cc`, (ovvero `stampa configurazione`) accetta solo la password.




A riga `00402e7e`, vediamo la chiamata di una stringa, ma *senza parametri*, nonostante il formato sembra richiederli. Non sono *fprint*, *sprintf* perchè richiedono precedentemente un puntatore a funzione o simili. Quindi è una `printf`. Tuttavia Ghidra non la interpreta bene, perchè non può non avere alcun parametro. La funzione è `__cdecl`, chi la chiama rimette a posto gli argomenti, perchè avendo parametri variabili, chi la chiama vede quanti ne ha chiamati. Vediamo inoltre il push di 20 byte, corrispondente a 5 parametri.

Rinominiamo la funzione della stampa, per indicare parametri variabili basta definirla con un argomento contenente "...".

Iniziamo a vedere i casi, prima di tutto dobbiamo disattivare la *protezione in tempo reale*, lanciando poi *Process Monitor* (analisi dinamica di base) e *Process Exporer*. Facciamo anche lo screenshot con *RegShot*. Lanciamo l'eseguibile da *shell* di comandi.

### Casi da analizzare

Nb: `pw` patchata, può essere qualsiasi cosa.

- `lab09_01 -in pw`, dopo averlo lanciato, fermo acquisizione di *Process Monitor*. L'eseguibile non si è cancellato. Non vengono creati nuovi processi dal nostro malware in *Process Explorer* (utile usare il filtro per nome, con nome del file). Facciamo `compare` tra le chiavi in Process Explorer. Ha aggiungo 117 chiavi, e modificate 88. Sono però chiavi di sistema, e manca la chiave vista all'inizio, che includeva `XPS`. Non abbiamo eseguito in modalità amministratore, La colonna `REPARSE` vuol dire che non sono riuscito a leggere la chiave. Dobbiamo rifare fotografie, monitoraggi, etc... Si richiude nuovamente. Ragioniamo: se fossi un malware, sarei interessato all'auto-attivazione nel sistema. Se riavvio il computer, il malware vorrebbe essere già pronto. Vedendo la differenza sulle chiavi, ci sono 15 chiavi aggiunte, tra cui quella `XPS` che stavamo cercando. Le altre chiavi aggiunte sono associate al nostro malware. Vengono inseriti dei `servizi`, equivalenti ai demoni di sistema in Linux. Viene espresso un percorso in cui è stato aggiunto il malware, diverso dalla posizione iniziale. Recandoci nella cartella, non lo troviamo. Questo perchè malware è 32 bit, sistema a 64 bit. Viene *mappato* in un'altra posizione, per trovarla bastare usare lo strumento di ricerca. Il nuovo path (generato automaticamente) è sito in `C\Windows\System\WOW64`. *Tasto destro* sul file, apri *percorso file*, è perfettamente identico all'originale, si è copiato! Sempre in `Process Monitor`, vediamo il **control set**, un insieme di configurazioni del sistema operativo, come è configurato attualmente, possono essercene diversi. Serve per tornare in vecchi control set, in caso di errore. Il malware tocca il control server corrente. Il malware sta cercando di diventare un *servizio*, dandogli un nome, di tipo Locale (non esclude che non possa esporsi sulla rete, ma generalmente possiamo vederlo come "*opera su questa macchina*"). Possiamo vedere i servizi del sistema mediante`services.mcs`, dove cerchiamo "Lab09-01", senza descrizione, ma con *stato empty*, cioè non è stato avviato, e c'è anche il *tipo di avvio* (automatico = 2, impostato in una delle 15 chiavi toccate, manuale, disabilitato).
  Finalmente abbiamo capito: questo tipo di avvio *infetta il sistema*, eseguendo l'**installazione**. Se aggiungessi il parametro opzionale, cosa toccherebbe? Probabilmente il nome del servizio, il nome del file, o il path di installazione.
- `lab09_01 -re pw`, rimuove il malware probabilmente. Facciamo sempre fotografia chiavi di sistema, usando *Regshot* prima del lancio e dopo il lancio. Troviamo 7 chiavi cancellate, che però non c'entrano nulla col nostro applicativo. Ha cancellato anche il *Services*, che non è più presente tra i servizi attivi. Ha tolto le chiavi di configurazione, non quella **XPS**. Lascia delle tracce. Bisogna differenziare il concetto di *chiave* e il concetto di *valore della chiave*.
- `lab09_01 -cc pw`, chiama una funzione che richiama *printf* senza argomenti. Ritorna una configurazione, con *sito* e *porta 80*. Esegue il **dump** della configurazione.

- `lab09_01 -c <a1> <a2> <a3> <a4> pw` probabilmente *setta* la configurazione. Proviamo a settare qualcosa e ad eseguire. Successivamente avviamo con flag `cc`, e vediamo che sono cambiati! Esegue **configurazione**.
- `lab09_01` è come parte il servizio, senza argomenti. E' il caso *normale*. Quindi non so *cosa faccia* il malware, perchè fino ad ora si trattava di configurazione.

Se in `services.mcs` troviamo in `XPS` la nostra configurazione, con stringa `30 31 00 ...` cioè le nostre stringhe, salvate nel registro di sistema.

**Analisi Ghidra - debugger parte 2**

**Casi analizzati precedentemente** Partendo dal file *patchato* (per superare la password), la func `004025b0` ottiene il nome dell'eseguibile, cioè " *lab09-01.exe*", poi invoca altra funzione interna, alla quale ho passato indirizzo su stack e dimensione pari a 1024, quindi è un buffer. La funzione è lunga, quindi vediamo sul *debugger*. Per capire meglio cosa fa una funzione, conviene guardare i parametri *prima* e *dopo*, tendenzialmente ci si sofferma su una operazione specifica. Infatti la funzione in esame è rinominabile come `getServiceName`, richiamata in `-in` ma anche nel flag `-re`, perchè devo sapere cosa eliminare.

Nel caso `-c` si esegue, passando `argv` coi rispettivi offset (riga sotto a `00402c79`), ed invocando `handler_c` (chiamato così da noi!).

Nel caso `-cc` è come abbiamo già visto, con la stampa di questi parametri. Manca il caso generale.

**Caso principale con nuove info** Nel caso *base*, viene chiamata `RegOpenKeyExA`, se esiste ritorna 1 e chiama un *handler*, se fallisce 0, e quest'ultimo caso porta al suicidio. Ovvero, si uccide se non è installato. Lui si accontenta della chiave, del servizio non c'è alcuna precisazione, quindi il malware funzionerebbe comunque. Vediamo la gestione dell'handler, al riferimento `402360`, nel caso corretto. Torniamo al *debugger*, mettiamo breakpoint su `402360`, cioè handler del servizio. Fallisce perchè l'ultima operazione è stata di tipo `-re`, quindi dobbiamo reinstallarlo. Adesso entra. Cosa ci aspettiamo? Sicuramente un *loop*, perchè se c'è un collegamento a qualcosa di remoto, c'è un processamento continuo.

# 23 novembre 2023

## Lab 09-01 [parte 4]

Dobbiamo cercare un *loop* senza fine, perchè in attesa di richieste. In `service_main` troviamo `handler_cc`, che otteneva dalla chiave il registro. Se avviamo il programma, senza parametri, ci aspettiamo un ciclo senza fine; tuttavia il programma pare si chiuda. Ma è effettivamente attivo? Se andiamo nella finestra dei servizi attivi, esso è registrato, ma non attivo. Se lo attiviamo da lì, abbiamo `errore 1053` troviamo un messaggio di errore che ci conferma il non corretto funzionamento. Per capirlo, dobbiamo tornare al debugger (*OllyDbg*) e capire dove si chiuda. Lo lanciamo *senza argomenti*, mettiamo un *breakpoint* su *main*. Attualmente abbiamo riferimento `00402360` per il service_main e `402AF0` per il main vero e proprio. Ciò ci porta alla funzione `00402eb0`. Il suo ingresso lo prende *stranamente* da `EAX`, non standard, quindi è simile ad una libreria. E' come se il compilatore fosse stato *costretto* a fare così. Mi serve studiarla? No, perchè non è lei il problema, ci sono passato attraverso. Successivamente, viene messo *forzatamente* 1 in `EAX`, indipendente dalla return della funzione precedente. Poi fa un `test` (viene fatto un AND bit a bit tra EAX e sè stesso, quindi salto se EAX è 0, quindi non salto mai alla funzione successiva) e `jmp`. Ciò ci suggerisce un ciclo. Un altro test ci porta a 402180, che è `handler_cc`, che possiamo ulteriormente analizzare. Essa chiama tre funzioni:

- `402e6a`, a cui passo la stringa `80/0` (buffer locale con stringa `80`, ultimo carattere è fine stringa), e viene eseguita una `atoi`.
- `401e60`, passo `http://www...` e il valore precedentemente convertito. (in basso a destra, in *OllyDbg*, vediamo i parametri passati). Essa invoca `401E60` con due argomenti, `19d2ec` e `400 hex` (1024 decimale). Su `EAX` ritorna 1 (lo vediamo in alto a destra di OllyDbg). Sotto c'è un controllo. Se tale controllo fallisce (variabile $\neq 0$) ritorna, sennò continua. Quindi è questo che ci sta bloccando. Soffermiamoci su questa.

Comunque, da `401e60`, vediamo svariate chiamate a funzioni, in cui si prende l'*hostname*, la *porta* e la stringa *qk6p...*. Se riavvio e la invoco, cambia! Ciò che produce è un `randonName`. Non è questa che dà problemi. Arriviamo a `00401AF0`, in cui passo 4 argomenti, ma non in tutti ci ho scritto dentro, quindi possono essere di output. Il ritorno è 1, è lei che fallisce. Vediamo cosa fa, mettiamo un breakpoint e ripartiamo, entrandoci dentro con `F7`: All'indirizzo `40165e` vediamo una call che Ghidra non riesce a risolvere. Per Ghidra, viene invocata `WS2_32.DLL::Ordinal_115`, ma non sa quale sia il nome simbolico. Sul *Web*, siamo fortunati che al primo link di Github ci sia un ragazzetto

che le ha numerate tutte! Se il ragazzetto non c'è, ce lo dice *OllyDbg*, è un `wSAStartup`, cher cercando sul sito Windows capiamo aprire una WinSocket per una connessione di rete. Ha successo? con `F8`, ritorna 0, funziona. Andando avanti, abbiamo `Ordinal_52`, cioè `getHostByName`, e funziona anche lei, perchè torna 0. Stiamo per eseguire `Ordinal_116`, cioè la chiusura della connessione. Allora è **fallita** `getHostByName`. Questo perchè ritorna un *puntatore alla struttura*, e solo se fallisce ritorna un puntatorenullo. Con *OllyDbg* e tasto `-` possiamo tornare al precedente valore dei registri. Non possiamo cambiare strada, ma vedere il passato. A `getHostByName` abbiamo passato la stringa `http://www.practicalmalwareanalysis.com`, ma esiste questo sito? Si. Capiamo meglio cosa fa questa funzione, usando il manuale Linux, in quanto quello Windows è in stato *To be Done*, cioè incompleto. Capiamo da `man gethostbyname` che ciò che dobbiamo passare è una *url*. Riconfiguriamo il malware: `lab09-01.exe -c ups www.practicalmalwareanalysis.com 80 60 AMW23`

Ora il puntatore è diverso da null. Riavviamo il servizio, e non funziona ancora. Continuiamo nell'analisi:
Troviamo la funzione `ntohs`: conversione dei numeri, perchè in un pc possiamo avere LittleEndian o BigEndian. Il manuale ci dice che converte dall'ordine di byte di rete TCP/IP all'ordine dei byte host (che è little-endian nei processori Intel). La connect funziona e si connette, stiamo più o meno intorno al riferimento `401835`. Andando avanti, troviamo a `4018e6` una `send`. Cosa mando?

- `GET` file random che abbiamo espresso prima, con getRandomName.
- `HTTP /1.0`

Si ottiene una `receive`, ci sono dei byte scambiati. Al riferimento `401F10`, vediamo che vengono passati *accenti gravi* ed *accenti acuti*. Viene poi invocata `403060`, che ritorna 0. Poi altre cose e ci ritroviamo in `EAX` 1.

Il fallimento si ha perchè la risposta che ottiene non è quella effettivamente fornita dla server. Non è il server giusto. Ma quale è? Boh. Creiamo noi il server! Deve stare in ascolto sulla porta 80 e ritornare la stringa richiesta. La VM manda messaggi a `wlan0:0:`, in cui viene specificato un indirizzo IP. Lo vediamo dall'host Linux. Potremmo mettere anche quello dell'interfaccia di rete. Cambiamo allora: `lab09-01.exe -c ups 192.168.74.2 80 60 AMW23` (l'indirizzo è quello del prof). Se pinghiamo tale indirizzo, funziona. Per il messaggio di risposta, usiamo `NetCat`, coltellino svizzero di *TCP/IP*. Meglio metterlo su Linux (per Windows è un malware), con:
`sudo nc -l -v -p 80`, in cui siamo in ascolto su tutte le porte. Rieseguiamo:
Abbiamo una connessione da indirizzo `192.168.74.100` e porta `60421`. Proseguiamo con `F8`, vengono fatte operazioni sulle stringhe, poi a arriviamo alla `send`, e vediamo sul terminale la richiesta del file. Ora, procedendo, rimane in attesa della risposta. Rispondiamo scrivendo sul terminale la sequenza di caratteri di accenti e prendendo invio.

A riferimento `401f1c` c'è il controllo su quello che si riceve, mettiamoci un breakpoint. In realtà, Windows vuole come terminatore di stringa "`CTRL-M``CTRL-J``CTRL-M`", cioè `\r\n`. (sul Terminale dovremmo vedere `^M^J^M`). Fatto sta che il programma è diffidente, e infatti da *Ghidra* vediamo che ci sono più stringhe controllate diverse volte. `FUN_403060` è complicata, ma in realtà è `strstr`, cerca nella stringa una sottostringa.
(serie di frasi in cui ci viene detto che ci vuole pazienza...)

Dalla funzione `402020`, vediamo comandi di `sleep`, `upload`, `download` etc... Come capiamo che si tratta di loro? Se vediamo i vari `DAT`, notiamo che sono delle stringhe brevi che non sempre Ghidra riconosce. Sono operazioni dal server verso il malware, ovvero possiamo fare anche operazioni di `cmd`, da remoto possiamo fare qualsiasi cosa. Tutto ciò unicamente instaurando un canale. Se non inviamo, dal server, uno dei comandi espressi, la connessione si chiude!

# 28 novembre 2023

## Comportamenti del malware vs debugger

Non possiamo analizzare tutti i meccanismi, in quanto essi sono numerosi e sempre rinnovati. Possiamo però lavorare con quelli più *consolidati*. Classifichiamo i comportamenti anti-debugging:

- **Detection**
  Il malware si rende conto di essere sotto debug, altera il suo comportamento. Può semplicemente non funzionare, o depistare l'analisi. E' diretto.
- **Identificazione del comportamento del debugger**
  Il malware non si rende pienamente conto di essere sotto debugger, ma nota dei pattern tipici. E' indiretto.
- **Interference**
  Il malware genera interferenze per le funzionalità di debug.

- **Exploit dei bug del debugger**
  E' una derivazione particolare del caso precedente, in particolare sfruttando alcuni bug del debugger stesso. Molto pericoloso.

## Detection

**ant1.exe**

In breve: `isDebuggerPresent` può controllare solo il processo *chiamante*.

Avviamo il programma `ant1.exe`, prima *senza* debugger, poi *con* debugger.

Nel primo caso, il programma mostra una finestra, con un commento.
Nel secondo caso, il programma termina subito.
Ciò richiede di usare un *disassembler*, che non è detto sia immune ad alterazione del comportamento.

Abbiamo visto che il programma, se lanciato a mano, mostra delle stringhe, quindi potremmo cercare quelle. Tali stringhe si trovano al riferimento `00402650`, in cui troviamo l'API `isDebuggerPresent`. E' una semplice funzione Windows.
**Una buona prassi è quella di vedere se tale funzione è presente ancor prima di avviare il programma, magari patchandola per non essere ostacolati.**
Tale invocazione è a `402666`, e quindi su **OllyDbg** mettiamo un *breakpoint* dopo esserci messi su tale riferimento (`ctrl+G`). Poi possiamo semplicemente forzare il bit booleano del check.

Si controlla solo il processo che esegue l'API.

**ant2.exe**

In breve: `CheckRemoteDebuggerPresent` può controllare qualsiasi processo su PC di cui passo handle.

Stesso comportamento. Apriamo con Ghidra, cerchiamo Windows → Symbols. Non c'è `isDebuggerPresent`, bensì `CheckRemoteDebuggerPresent`. Non vuol dire che il debbuger sia remoto!

---

Ugualmente, andiamo su **OllyDbg**, puntiamo a riga `402676`, mettiamo breakpoint, eseguiamo con F9 , fissiamo `EAX = 0` (prima è a 1) e possiamo procedere.

**ant3.exe**

In breve: `NTQueryInformationProcess` usa `ProcessDebugPort` che ritorna 1 se il debugger è presente. `NTQueryInformationProcess` caricata a runtime con `LoadLibrary` per poi prendere l'indirizzo della funzione con `getProcAddr`.

Fino ad ora siamo stati faciliati dal fatto che ci fossero delle chiamate a funzioni palesi. Apriamo con Ghidra, cerchiamo i simboli. Le due funzioni di prima *non* sono presenti. Un'altra possibilità è data da `NtQueryInformationProcess`, API di basso livello.
Le `.dll` di Windows presentano tre livelli:

- Le `.dll` di alto livello interagiscono direttamente con l'applicazione. Ne è un esempio `USER32.dll`. Queste utilizzano `.dll` di *medio livello*.
- Le `.dll` di medio livello presentano nomenclatura del tipo "`Nt...`".
- Le `.dll` di basso livello presentano nomenclatura del tipo "`Zw...`".

`NtQueryInformationProcess` non è ben documentata, perchè non usata direttamente dal programmatore. Tra le varie informazioni che gestisce, c'è il parametro `[in] ProcessInformationClass`, che include l'informazione `ProcessDebugPort`, che ritorna 1 se stiamo usando un debugger. Tale funzione non è però presente. Quindi o viene usata in modo diverso, o viene usato altro. Infatti, la troviamo tra le stringhe, non tra le funzioni. *I simboli che trova il debugger sono trovati al momento della compilazione*, allora tale API è invocata con un altro sistema. Ciò ha senso perchè, come abbiamo detto, raramente viene invocata direttamente. Il riferimento che troviamo è `Ntdll.dll`, con `loadLibraryA` la carichiamo a **runtime**, in essa sarà presente `NtQueryInformationProcess`; con `getProcAddress` prendiamo l'indirizzo di partenza di tale funzione. Possiamo bypassarla in vari modi: patch del flag, uso di `NOP`,... .

#### ant4.exe

In breve: Usa `OutputDebugString()`, se non c'è debug, ritorna un errore, diverso da un errore precedentemente settato con `setLastError` e preso con `getLastError`. Se errore vecchio e nuovo sono uguali, c'è debugger.

Funziona solo su *Windows7*. Qui si ha `OutputDebugString()`, la quale permette di inviare una stringa al debugger, invece di usare una `printf`. Se il programma non è debuggato, *prima* resistuiva un errore in modo *subdolo*, cioè mediante `getLastError`, ovvero il codice dell'ultimo errore; oggi `NULL`. Nel programma, basato su vecchia implementazione, prima si imposta un errore generico con `setLastError`, poi `OutputDebugString()`, e poi `getLastError`:

- Se l'errore è lo stesso, vuol dire che non c'è stato alcun errore nel passare una stringa al debugger, quindi c'è il debugger di mezzo.
- Se l'errore è diverso, c'è stato un errore nel passare la stringa al debugger, ovvero non c'è il debugger.

### Come funziona un processo in Windows

Tutte le informazioni del processo, ovvero istanza del programma in esecuzione, sono contenute in un'area kernel.

In Windows, queste informazioni non sono tutte nel kernel, una percentuale rilevante di queste strutture dati è presente nella zona user.

La struttura dati **Process Environment Block** è usata dal S.O. per memorizzare informazioni rilevanti per il processo. Ogni processo ha la propria **PEB**. Tale struttura presenta molti campi *reserverd* (leggasi: Windows non ci dice cosa sono, anche se molti sono stati compresi, sicuramente chi scrive malware sa a cosa servono!).

Il **PEB** è specifico di Windows e si concentra sull'ambiente del processo, il PCB è un concetto più generale e fondamentale per la gestione dei processi nei sistemi operativi, fornendo informazioni sullo stato e le risorse del processo.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged; //Ci interessa lui!
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  //... altri parametri a cui non siamo interessati..!
} PEB, *PPEB;
```

Notiamo subito `BeingDebugged`, che ci dice se siamo debuggati o meno.

#### Come arriviamo alla PEB?

Windows, come Linux, è piatto, non usa i segmenti. Gli indirizzi che vediamo, sono offset di segmenti che partono a 0, quindi **gli offset sono indirizzi lineari**. Ogni thread attivo di un processo ha una struttura dati, accessibile all' offset 0 utilizzando il registro segmento `FS`. La segmentazione è comunque sempre attiva, e `FS` presenta le informazioni specifiche di quel thread.

All'offset `0x30` di `FS` si ha un pointer alla `PEB`. La *struttura* PEB è *unica* per il processo, ma ogni processo può avere moltissimi thread.

Quindi dobbiamo puntare a `FS:[30h]` e poi *spiazzarci al secondo campo* `BeingDebugged`, ovvero ci spiazziamo di `0x02h`.

In linguaggi orientati agli oggetti, c'è un modo per organizzare i blocchi `try & catch`, e tale meccanismo, che cattura le eccezioni *asincrone* rispetto l'esecuzione principale, sfrutta `PEB`, quindi può essere usato anche per cose legittime.

#### ant5.exe

Sfruttiamo le conoscenze appena acquisite, senza uso di API:

```
MOV EAX, FS:[0x30]
MOV EAX, dword ptr [EAX + 0x2]
```

### Controllo dell'Heap

Tra i vari campi di `PEB`, all'offset esadecimale `0x18`, abbiamo riferimento alla struttura `ProcessHeap`, cioè come gestisco la parte di memoria per l'allocazione dinamica. Prendo da qui le pagine.

---

**Nota**: Il parametro associato all'Heap dovrebbe essere il primo campo associato a `Reserved4`.

---

Il processo, se debuggato, ha una **gestione dell'heap diversa** rispetto al caso non debuggato. In particolare, vengono coinvolti i flag `ForceFlags` e `Flags` aventi:

- valori 0 in assenza di debugger.
- valori $\neq 0$ in presenza di debugger.

Il problema è che tali flag cambiano posizione di offset in base alla versione di Windows.

| | **WinXP** | **Win10** |
|---|---|---|
| **Forceflags** | `0x10` | `0x44` |
| **Flags** | `0x0c` | `0x40` |

## Identifying

### Ricerca delle chiavi di registro

Idea: *controlla* le tracce lasciate da debugger, come chiavi del registro.

Il malware potrebbe controllare le tracce che un debugger lascia sull'Os quando installato, magari controllando le chiavi del registro.
Esempio di chiave:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\Currentversion\AeDebug`

Configurata se è presente almeno un debugger, di default punta a *doctor Watson*, che è un debugger incluso in Windows.

### Analisi finestre aperte

Idea: tramite `FindWindowA` ritorna handler del processo con nome di una finestra.

Un altro metodo è trovare la finestra che si apre quando si lancia un debugger. Alcune API, come `FindWindowA`, restituiscono un *handle* al processo avete un certo nome `name` della finestra.

### Scan della memoria

Idea: Controllare in memoria la presenza di `INT3`.

Sappiamo che, nei debugger, si usano i *breakpoint* con `INT3 0xCC`. Un malware potrebbe fare una scan della memoria. Possono esistere falsi positivi, ad esempio sui dati non è un problema, ma sugli operandi sì! Se i controlli vengono fatti sul software, possiamo usare debug hardware, o viceversa. Se vengono controllati entrambi, no!

Un'altra variante, poco usata, sarebbe usare `INT3 0xCD 0x03`, che si comporta nello stesso modo.

Prendiamo questo snippet scritto dal prof:

```
CALL $+5        # Metto su stack indirizzo di ritorno della funzione e salta avanti di 5 byte.
POP EDI         # Memorizzo in EDI il valore in cima allo stack,
                  Ovvero il valore di ritorno della funzione.
SUB EDI,5       # Sottraggo 5 ad EDI, torno indietro di 5 byte.
MOV ECX, 0X400 # dimensione codice messa in ECX, pari a 1024.
MOV EAX, OXCC  # mettiamo CC (opcode di INT3) nel byte AL, appartenente a EAX.
REPNE SCASB     # debug, operazione su stringa, confronta la locazione puntata da EDI e byte AL
                  Ripete il confronto fino a quando il flag ZF è 0, o ECX diventa 0.
                  Serve per cercare un byte (ad es. 0xCC) in una area di memoria.
```

```
JZ debuggerDetected  # Se il flag ZF viene impostato dopo il confronto (ovvero se ho trovato il byte)
                       saltare a questa etichetta.
```

Sostanzialmente con `repne` cerchiamo la stringa `CC`.

#### Timing checks

Idea: vedere il tempo di esecuzione tramite `rdtsc`, se troppo alto, qualcuno ha fermato l'esecuzione per l'analisi.

Prendo il tempo di esecuzione di una porzione di codice. Se questo tempo risulta alterato (ad esempio mi fermo su una parte di codice per ragionarci), allora il malware capisce che c'è qualcosa che non va. Il modo in cui viene realizzato è nascosto, non usa API di Windows, bensì usa `rdtsc`, che legge il *Read Time Stamp*, contatore aggiornato con frequenza molto elevata. Attualmente, non è più correlato alla frequenza del bus di sistema (es: 1 $GHz$, anche perchè oggi sono molto variabili). Il valore ottenuto è a 64 bit, e quindi richiede i registri `EDX:EAX`.

Uno snippet di esempio è:

```
RDTSC            # prendo tempo
MOV ECX, EAX     # metto tempo in ECX
...              # possibili altre operazioni
RDTSC            # prendo nuovo tempo
SUB EAX,ECX      # differenza tra tempi campionati
CMP EAX, 0XFFF   # confronto con un valore fissato
JB NoDebugger    # se abbastanza piccolo, no dbg
DebuggerDetected # altrimenti, è presente dbg
```

Si può anche usare una interfaccia per il *Time Step Counter*, ad esempio con `QueryPerformanceCounter`, usata anche nell'esecuzione iniziale di un programma. Se venisse usata *non all'inizio*, il malware potrebbe insospettirsi.

Il *Time Step Counte*r non è un tick, bensì è legato ad eventi sui registri. Il *Tick Count* è invece legato ai tick, e c'è l'API `GetTickCount`, se la durata di tempo da misurare è $<$ tempo del tick, non è utile usarlo.

### Interference

Idea: Il malware mette `INT3` dove lo metterebbe il programmatore, chi prende il controllo per primo, vince!

Il debugger usa breakpoint, software o hardware, ma chi ci dice che anche il malware non possa usare l'eccezione `INT3`? L'idea è che *vince chi ha il controllo per primo*, quindi chi crea malware deve anticipare l'analista, e quindi deve pensare *dove* l'analista potrebbe mettere il breakpoint. Il caso base è, ad avvio debugger, mettere un *breakpoint* sull'entry point. Ma **non è vero che l'entry point è la prima istruzione in assoluto**, la parte precedente è di *inizializzazione*, che può essere definita dal programmatore (tralasciando la parte gestita dal sistema operativo). Quest'area grigia lasciata al programmatore è associata al concetto di **Thread Local Storage callbacks**. L'area *TLS* è locale ai thread, ma deve essere inizializzata, mettendo del codice in una particolare sezione `.tls`, dove anche il programmatore (e quindi il malware) può metterci le mani.

La soluzione sarebbe fermarsi *prima dell'inizializzazione del processo*, ma non tutti i debugger è possibile fare ciò. **OllyDbg** permette di fare ciò.

In `Options (alt+D)` →`DebuggingOptions`→`Events` → `Make First pause at.. System Breakpoint` possiamo fare ciò.

**Ghidra** carica anche le sezioni *TLS*, ma dobbiamo andare esplicitamente a vedere le sezioni *TLS*, sennò lui parte dall'entry point. Altra zona delicata è rappresentata da *costruttori* e *distruttori*, in particolare i *costruttori* sono collocati dopo l'entry point e prima del main.

#### Interrupt software ed eccezioni

Per interferire col debugger è sufficiente generare **eccezioni**, come `0XF0`, che non fa nulla di particolare, infatti il debugger la rigira al programmatore, che a sua volta la ripassa al debugger, rallentando l'analisi. Con 1000 eccezioni la cosa è pesante, per questo, molti debugger permettono di evitare questo ciclo, magari andando avanti senza fermarsi.

In OllyDBG c'è una sezione in `DebuggingOptions` in cui scegliere le eccezioni da skippare. Troviamo anche `INT3`, ma non dovrebbe essere attivo visto che lo usiamo? OllyDBG riconosce le sue `INT3`, generate con lui, da altre non generate da lui. Saranno proprio quest'ultime ad essere ignorate. Tanti altri debugger non riconoscono questa differenza, puntando sul fatto che un normale programma usa molto poco `INT3`. Da qui possiamo ignorare anche `0xF0`.

**NB**: Anche la divisione per 0 viene ignorata, in quanto modo molto semplice per generare eccezioni. C'è una differenza tra fare:

- `int v = 3/0` , che è *diretto*, a compile time.
- ```
  int v = 0
  ...
  w = 3 / v  // a runtime
  ```

In quest'ultimo caso il compilatore non segue il flusso dei valori assunti da v, portandolo forzatamente ad eseguire l'istruzione `div` (istruzione di divisione piuttosto lenta), generante l'eccezione.

**Se ci fosse INT3 nelle istruzioni del programma?** Non tutti i debugger riescono a differenziarli da quelli introdotti durante il debug. Un debugger nato male è **WinDbg**, poichè inadatto per analizzare i malware.

Un flusso normale del programma è:

```
ASM1 // istr1, solitamente eseguite in sequenza a meno che
ASM2 // qualche istruzione generi eccezione, allora controllo
ASM3 // passa al SO, che a sua volta la passa al debugger, se presente.
```

Magari durante `ASM2` viene generata un'eccezione `CC`, passata a **WindDbg**, che viene girata all'analista. Qui possiamo andare avanti passo passo o fermarci.

Abbiamo visto che un'altra eccezione simile a `INT3` è `CD03`, composta da 2 byte piuttosto che uno (come `CC`). WindDbg non lo sa, e crasha.

Altra istruzione è `INT 2D`, livello Kernel, ma comunque concettualmente uguale a `INT3`.

In **Intel** esiste anche `OXF1` ovvero `IceBP`, di tipo hardware. L'istruzione successiva a questa viene eseguita senza che il programmatore possa *fermarla*. Quindi scappa al suo controllo. E' scarsamente documentata.

## Interrupt IRQ vs Exeception

La differenza è che:

- **Interrupt** è *asincrona* rispetto l'esecuzione, dipendente da dispositivi esterni, come pacchetti IP o altro.
- L'**exeception** è *sincrona* rispetto all'esecuzione, legato alle istruzioni macchina, come dividere per 0, ovvero nel codice che sto eseguendo. Oppure un *page fault*.
  Servono per implementare meccanismi di memoria dinamica, o *copy & write*.
  Sono però anche utili per il *programmatore*, che può gestirle (se le ha previste, come la divisione). Esistono 3 casi:
  - *Eccezione ignorata*: come se non fosse mai avvenuta.
  - *Eccezione che genera abort*: il processo viene ucciso, come nel caso della divisione.
  - *Eccezione catturata*: la gestisco io.

**Il flusso dell'istruzione macchina non viene eseguito in sequenza**, sia perchè c'è il meccanismo di sorpasso **OOO** (speculazione software), sia perchè possono esserci fattori hardware come **interrupt**.
Solo a livello *logico* questa visione ha senso.

Se prendiamo `MOV EAX, [EBP]`, questa istruzione non è sempre detto che si svolga normalmente, ad esempio `EBP` potrebbe contenere un indirizzo invalido, e quindi richiedere più tempo del dovuto.

Oppure, con una `malloc` richiedo 100 pagine, ma non mi vengono *fornite* appena le richiedo, me ne verrà data una solo quando vi accederò (verrano materializzate quando vi accedo). La pagina *fisica* mi viene fornita solo in questo momento.

### Structured Exeception Handling SEH

Ogni compilatore estende il linguaggio, con elementi esterni al linguaggio.
Nel caso dei compilatori Microsoft, parliamo di una variante di C o C++, che chiamiamo `C-variant` e `C++ variant`.
Il meccanismo `SEH` è stato introdotto nei sistemi **Borland**.

**NB: i try-catch possono essere ricorsivi, e quindi annidati.**

Come la riconosco?
```

### C - variant

```
void foo() {
    try {
       //codice che potrebbe generare eccezione
    } except (...) {
        //gestione
    }
}
```

E' possibile anche trovare blocchi `finally {...}`.

### C++ base

```
void foo() {
    try {
       //codice che potrebbe generare eccezione
    } catch (...) {
        //gestione
    } catch (...) {
        //gestione
    }
}
```

Qui abbiamo usato delle `catch`.
Vediamo adesso la variante.

### C++ variant

```
void foo() {
    __try {
        //codice che potrebbe generare eccezione
    }
    __except (...) {
        //gestione
    }
}
```

Nel caso del codice C++ standard, utilizziamo `try`, `catch` per gestire l'eccezione. Nel caso del codice SEH, utilizziamo `__try` e `__except` per gestire l'eccezione.

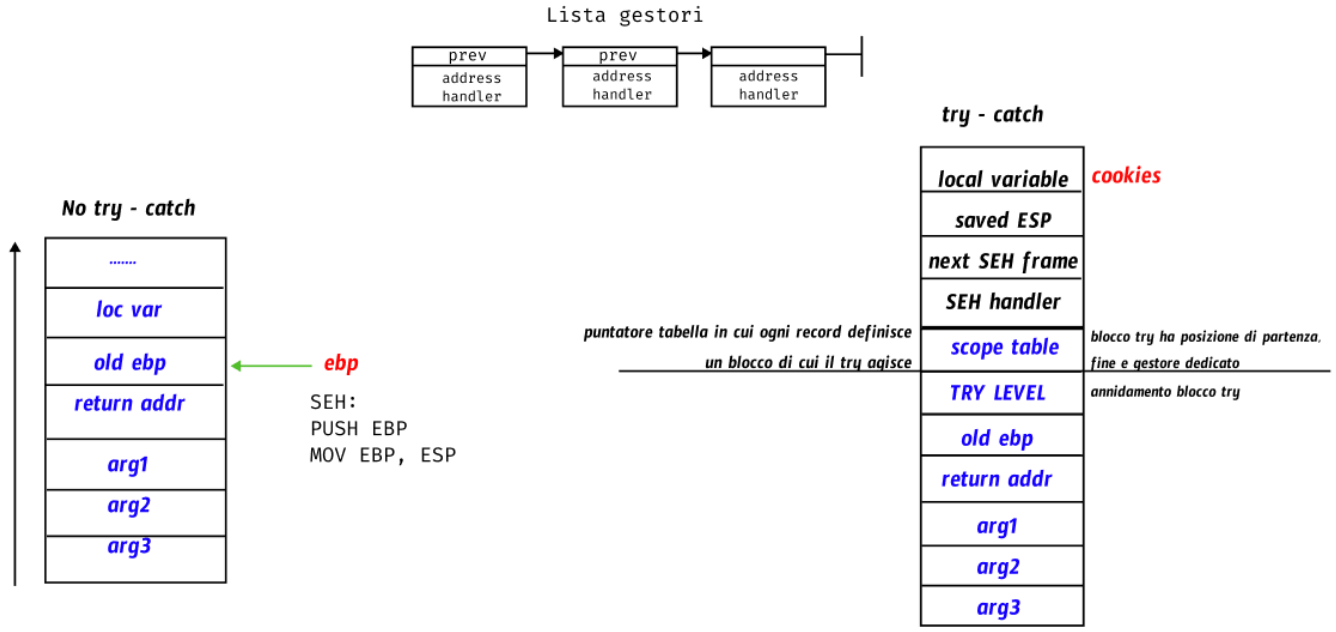
In breve, *Structured Exception Handling (SEH)* in Windows offre un approccio strutturato per gestire eccezioni e errori di runtime. Tuttavia, essendo specifico di Windows, può compromettere la portabilità del codice. Spesso utilizzato in contesti a basso livello o critici del sistema operativo (quindi caratteristiche proprie di Windows, o driver specifici), SEH offre un controllo dettagliato sulle eccezioni.

C non ha un sistema integrato per la gestione delle eccezioni come C++.

## Digressione su Stack

Ricordiamo che con `EBP` salviamo la testa dello stack. Dobbiamo ricordare la ricorsività delle eccezioni nello stack.

Manteniamo una lista dinamica di gestori (nell'immagine, sulla destra), con un campo `prev` che punta al precedente. Il più "vecchio" non ha un campo `prev`. Nel gestore manteniamo anche l'*indirizzo dell'handler*. Sono sullo stack, e ci mettiamo l'elemento corrente di questa funzione. Sullo stack ci va anche un counter che ci dice il livello di annidamento, per capire se è necessario annidare o fermarsi.




Esistono almeno 5 varianti di **SEH**, questa che vediamo è la variante 3, in quanto le precedenti non più usate, la versione 4 aggiunge i *cookies* (ovvero, un tipo di attacco è sovrascrivere gli indirizzi di ritorno, cioè saturare lo stack alla cieca; una possibile *patch* consiste nell'inserire un valore predefinito scritto ad inizio funzione e controllato prima di sfruttare il return address; tale protezione è definita come *cookies*) e la versione 5 è object-oriented.

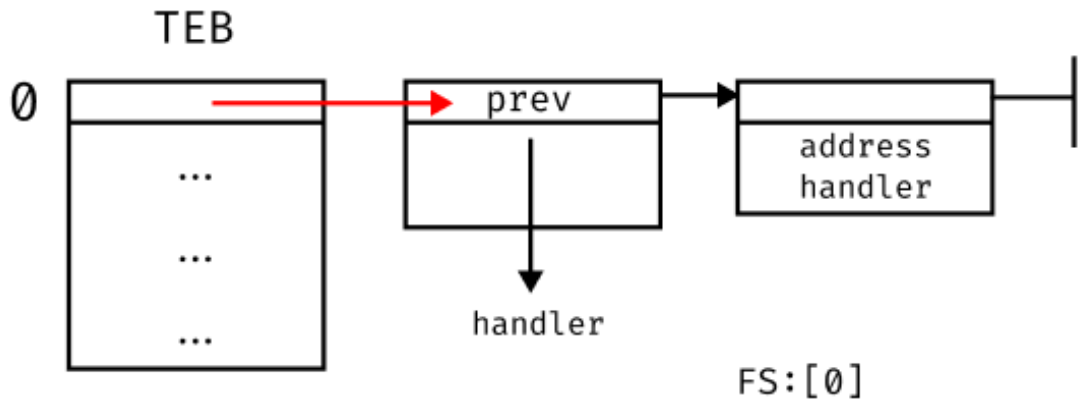
*Se arrivo in fondo e non trovo un handler adatto?*

Il tutto viene mandato al gestore di default, che non la gestisce, e fa morire il programma. Tuttavia posso cambiarlo, tramite API `SetUnhandledExeceptionFilter`.

**NB**: ogni thread ha il proprio stack, allora la lista è per ogni thread.

### Dove si trova la testa della lista?

Si trova nel **Thread Environment Block**, in cui ad offset 0 abbiamo puntatore al primo elemento per la gestione dell'handler. Come ci arrivo? `FS:[0]`.




Con le seguenti istruzioni macchine, registro un nuovo Exeception Handler:

```
PUSH ExceptionHandler
PUSH FS:[0]
MOV FS:[0],ESP
```

Il valore di `ExceptionHandler` sarà nello stack *sopra* il valore attuale di `FS:[0]`, quindi `ExceptionHandler`, essendo nella testa dello stack, viene puntato da `ESP`.

Semplicemente metto nello stack tale handler, e lo sposto in `FS:[0]`, in quanto è l'unico posto *certo* in cui posso metterlo, ovvero in testa, perchè annidato. Parto dal più interno, ad uscire.

# 5 dicembre 2023

Abbiamo lista di gestori di eccezioni installata. E' una lista poichè sono innestabili, ovvero abbiamo blocchi annidati l'uno dopo l'altro.

#### anti6.exe

Se lanciato da solo, funziona senza problemi. Aprendolo con **OllyDbg**, viene mostrata la finestra di avvio. Sembra funzionare. Andando avanti, però, il debugger mostra un errore associato a qualche indirizzo di memoria, e non sa come continuare l'analisi. Nel codice sorgente (qui per semplicità didattica lo vediamo) si ha:

```
xor eax, eax          #azzera eax
push L0               #etichetta qualunque
push DWORD PTR FS:[0]
```

Questo porta alla creazione di una nuovo **handler di eccezioni**, inoltre `FS:[0]` è tipica di un blocco di Structured Exception Handling (SEH) in ambiente Windows. Il valore a `FS:[0]` rappresenta il puntatore al blocco di registrazione delle eccezioni corrente. Nello specifico, lo completiamo con:

```
xor eax, eax
push L0
push DWORD PTR FS:[0]
INT3
INC eax
L0: TEST eax,eax
    JNZ debugger
```

Quando tale codice viene eseguito, e non c'è il debugger, `INT3` genera eccezione passata direttamente al programma. Il programma vede il primo gestore dell'eccezione, vede dove punta, ovvero in `TEST eax,eax`, cioè saltando l'**incremento**. Il programma crede di essere il gestore dell'interruzione. Il gestore ritorna 0 e 1 generalmente, però questo vale per un vero gestore, e qui il ruolo del gestore è del programma. Nelle opzioni di OllyDbg, in `debugging options`, togliamo la spunta sulla eccezioni `INT3`. Se avviamo, ci fermiamo proprio sull'istruzione `INC eax`, e dopo termina. Togliendo la spunta, la misura *antidebugger* del programma funziona. OllyDBG è un po' più furbo e quindi ci rendiamo conto di questa contromisura. Il programma mostra l'errore perchè si comporta come gestore delle eccezioni, in un contesto diverso da quello normale. OllyDbg non è pronto per queste cose.

Se avviamo il programma, facciamo run con `F7`, arrivati a riga`0040267B` vediamo che il malware mostra una schermata, in cui ci rivela di aver capito di essere in un debugger, rompendo i comandi del debugger. Infatti, *il trap flag è stato riazzerato e il malware eseguito.*

Quindi, anche con esecuzione in maniera controllata, il malware può sfuggire dal nostro controllo. Come fare ciò dipende dal malware e dal debugger. La sequenza di istruzioni descritta prima è un campanello di allarme.

## Exploiting vulnerabilities

**Premessa: Il debugger è un programma, non è esente da bug.**

OllyDbg è in versione 1.10, **mai** usare versioni precedenti!

#### PE header bugs - anti7.exe

*Idea*: Una delle tecniche che si utilizza, o meglio che è stata usata nel passato, è quella di modificare la testata PE del file eseguibile in modo che continui ad essere eseguibile per Windows ma mandi in crash OllyDbg. Per modificare la testata PE si può usare un tool specifico, come *PE Bear* in modo da inserire nella testata PE ciò che si vuole.

Apriamo con `PE-bear`, per vedere la testata PE di un file. Analisi statica, non sto eseguendo il programma. Tra i vari campi di PE, possiamo definire vettore con descrittori **RVA**. La lunghezza di tale vettore nel PE, in questo esercizio, è di 16 (che è generalmente il valore massimo possibile), e lo vediamo in *number of RVAs and Sizes*. Che succede se lo modifico? Creiamo una copia del file eseguibile (che sarebbe il file **anti7a.exe**), avente dimensione del vettore $ 7FFFFF$, molto grande, sicuramente più di 16.

Se lo eseguiamo, il programma funziona. Al sistema operativo non interessa, fa *il minimo tra questo valore e* 16*, e poi lo esegue*. Se lo apriamo con *OllyDBG*, inizia a scrivere più del necessario, sovrascrivendo altre aree del codice.

Quindi se apriamo **anti7a.exe** con OllyDbg, fornisce un codice di errore, ma poi esegue.

Altro aspetto è legato alle sezione *Section Header* di `PE-BEAR`.

Ci sono varie dimensioni, tra cui `raw size` e `virtual size`. **Non sempre coincidono**. Ad oggi prendono il *minimo* tra i due. In versioni precedenti veniva presa direttamente la `virtual size`. Noi possiamo modificare i campi, se il valore di `raw size` > `virtual size`, Pe BEAR me lo fa fare, ma me lo segna come errore. Il problema lo ha OllyDbg.

Versioni precedenti di OllyDBG non comparavano `raw size` e `virtual size`, vedevano direttamente `raw size`, e se sbagliata, *portavano al crash.*

#### Stringa mal processata - anti8.exe

Funziona bene senza debugger.

Apriamo OllyDBG, e lo lanciamo. OllyDbg crasha. E' un *bug* non fixato.

La vulnerabilità è in `OutputDebugString(msg)`, che manda stringa da stampare a debugger che sta debuggando il processo. Se il messaggio è `msg = %s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s` va in crash. Questo perchè il debugger interpreta la stringa in questo modo: Sullo stack c'è il puntatore a `%s`, sotto c'è un altro puntatore, sotto un altro, fino a quando arriva ad uno 0, arrivando a **segfault**. Cioè si fida che io passi tutti *puntatori validi*, quando trova il puntatore a 0 (o qualsiasi puntatore particolare) si rompe.

Il programma che lo rompe, è semplice:

```
int main()
{
    OutputDebugString(%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s);            //stringa malevola passata
    MessageBox(NULL, "Debuggah?\n", "Nooh non debugga!", MB_OK);   //finestra a schermo con msg
    return 0;
}
```

Su Win10, `OutputDebugString` è usabile per capire se c'è il debugger.

## Misure anti-disassembler

#### AntiDis.exe

Eseguiamo con OllyDBG, e va in crash. Apriamo con Ghidra, e cerchiamo in `windows` i riferimenti alla stringa `OutputDebugString` :

al riferimento `401540`, a riga `00401537` , c'è jump ad etichetta in rosso, e il *salto non riesce* poichè non trova l'indirizzo `00641a03` nella memoria del programma. Succede anche sotto nel codice.

In un debugger semplice, si trova l'entry point e si interpretano le istruzioni. Se la lunghezza non è fissa, devo sapere dove inizia e finisce l'istruzione.

Se prendiamo il function graph, i blocchi di codici sono legati tra loro tramite *jump*.

#### Digressione sui jump

1. **JMP (Jump)**: Questa istruzione viene utilizzata per saltare *incondizionatamente* a un'altra parte del programma. Non fa alcun controllo sullo stato dei flag del processore o sui dati, ma semplicemente trasferisce il flusso di esecuzione a un'etichetta specificata.
2. **JZ (Jump if Zero)**: Questa istruzione si riferisce al flag "Zero" nel registro dei flag del processore. Se il *flag Zero è impostato* (ovvero, se l'ultima operazione ha restituito un risultato zero), allora si verifica il salto alla posizione di memoria specificata.
3. **JNZ (Jump if Not Zero)**: Al contrario di `JZ`, questa istruzione salta a un'altra parte del programma solo *se il flag "Zero" non è impostato*. Se l'ultima operazione ha restituito un risultato diverso da zero, questa istruzione effettuerà il salto.

Preso salto condizionale `jz l0`, esso può essere preso o meno.

Un disassemblatore flow-oriented analizza il flusso, ad esempio una funzione, e trova una biforcazione:

- seguire l'istruzione `q`, ovvero quella subito dopo `jz l0`.
- andare a disassemblare dall'indirizzo `l0`.

```
  xor eax,eax
  jz l0
q: ...//do something

l0: ...//do something else
```

Ghidar fa entrambe, mettendo prima `l0` e sopra `q` sullo **stack**. Di solito, i disassembler *prima* continuano a disassemblare `q`, poi si ricordano che `l0` è l'inizio di una istruzione macchina (salvata prima sullo stack) e quando avranno finito con `q` , prenderanno dallo stack `l0` ed analizzeranno lei. Il tutto ricorsivamente.

La priorità è quindi data all'istruzione dopo il jmp.

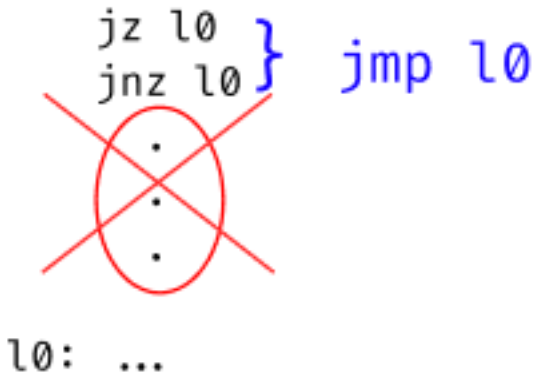
Se il programma proseguisse cosi:

```
  xor eax,eax
  jz l0
q:
  jnz l0
r: ... //do something

l0: ...//do something else
```

Riferendoci alla stesse idee di prima, mettiamo `l0` sullo stack, continuiamo leggendo `q`, mettiamo nuovamente `l0` sullo stack (perchè è ciò che facciamo in `q`) e poi proseguiamo con `r`.

Il problema è che, mettendo questi due jump insieme, a **runtime**, equivale a `jmp l0`. Ghidra sbaglia perchè cerca di seguire il flusso in `r`, visto che tale percorso non verrà mai preso in quanto, grazie alle due jump appena descritte, salteremo sempre, e quindi non c'è bisogno di vedere "cosa c'è sotto", in quanto non ci andremo mai.




Sbaglia perchè non è in grado di dire che questi due salti condizionali equivalgono ad un unico salto conzionale.

Se avessimo messo direttamente `jmp l0`, sotto (dove è cerchiato in rosso), avrebbe messo dei punti interrogativi, dicendo "non so se qui dentro ci salta qualcun altro!"

Tuttavia, vedendo i due salti condizionali, vede quello che c'è dopo (cerchiato in rosso), come un'istruzione, e quindi disassembla, sbagliando.

Stiamo **disallineando** il codice macchina.

**Esempio:**

```
xor eax,eax
jz l0
```

Azzera $x$ e salta se è 0, per noi è uguale a `jmp l0`. Ghidra però va sotto solo in questo primo caso, in questo secondo caso no:

```
    xor eax, eax
    jz l0
    .BYTE e9 //Tale istruzione è un byte "nascosto".
             //L'opcode e9 e' un salto incondizionato (JMP)
l0: POP eax
    RET
```

Azzero `eax`, salto se è 0, cioè salto ad `l0`, faccio `pop eax` e `return`.

Il disassembler comincia ad interpretare `e9`, vede i byte di tale istruzione (5 byte), prende `pop eax` e `RET`, portandolo a fare un salto ad un indirizzo sballato, in quanto lo stesso byte è visto diversamente:

- Il processore vede salto incondizionato (`e9`)
- Il primo byte `POP` lo vede come codice operativo e non un dato.

Nel codice, si salta a `LAB_00401537+1` (avendo doppia istruzione JZ e JNZ), +1 vuol dire che sta **saltando all'interno di un'istruzione macchina**, sotto c'è `00401537`, con byte `e9 c6 04 24 00`:

Il primo byte `e9` è associato ad una *call*, facendo +1 dovrebbe saltare al secondo byte `c7`. Quindi stiamo facendo una call `e9` usando `c6 04 24 00` come indirizzo a cui saltare. Noi questa call non la **eseguiremo mai!**

Azzeriamo questa errata interpretazione, facendo *clear code bytes* (tasto `c`), saltiamo direttamente al `lab_00401538`, lo selezioniamo tutto, e facciamo `disassemble`.

Immagine di esempio degli anni precedenti (riferimenti diversi!):

```
004026bb 74 03          JZ                  LAB_004026bf+1
004026bd 75 01          JNZ                 LAB_004026bf+1
                    LAB_004026bf+1                          XREF[0,2]:
004026bf e8 e8 6b       CALL                SUB_002e92ac
         ee ff
```

**Come si scrive codice del genere ad alto livello?**

Esistono macro che sporcano il tutto.

```
#define OBSCURE_1 \
    __asm__ __volatile__("jz 1f\n\t" \
                         "jnz 1f\n\t" \
                          ".byte 0xe8\n\t"\
                          "1:");

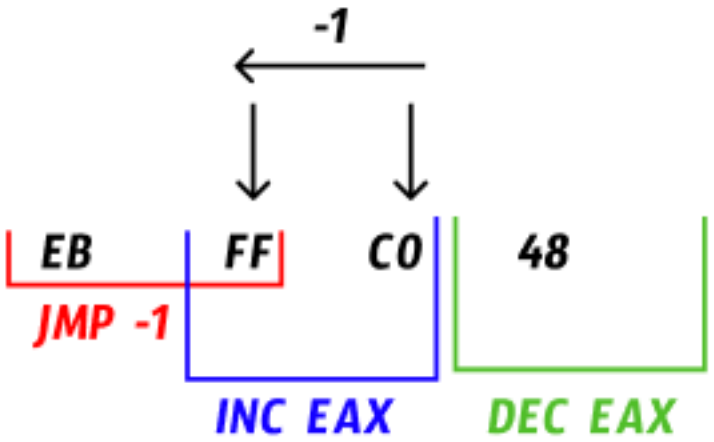
#define OBSCURE_2 \
    __asm__ __volatile__(\
                         "jz 1f\n\t" \
                          ".byte 0xe9\n\t"
                          "1:" : : "x"(0) : );
```

# 7 dicembre 2023

## Salti condizionati e non

Riprendiamo il concetto del salto condizionato, questo può essere preso oppure no, il disassemblatore è *flow oriented*, man mano che analizza il flusso si rende conto che c'è una divergenza nel flusso, in quanto può andare all'indirizzo `l0` che segue la `jz`, oppure andare su `l1`. Il disassemblatore ha al suo interno uno stack, mette entrambe sullo stack e continua a disassemblare *cosa c'è dopo*. Una volta finito il flusso, passa all'altro salvato e non ancora analizzato, ricorsivamente. Il proseguire può contenere una istruzione `jnz l0`, quindi di nuovo si mette `l0` nello stack e prosegue. Ma non è un vero salto condizionale, perchè fare `jz` e poi `jnz` è equivalente! Da un punto di vista di codice eseguito a run time, equivale a `jmp l0` e quindi il disassemblatore sbaglia nel cercare di seguire il flusso perchè non è in grado di dire che i due salti condizionali sono un unico non condizionale.

Quindi, in breve: Quando c'è `jmp l0`, salta ad `l0`, sotto a `jmp l0` non sa cosa c'è, e non azzarda. Se facciamo due jump `jz z0` e `jnz l0`, per il disassembler c'è una biforcazione. Quindi *interpreta ciò che è sotto come se il processore andasse in sequenza*, ma è falso, perchè in realtà saltiamo (essendo due condizioni opposte, una sarà vera per forza). Quando abbiamo un salto condizionale, potremmo analizzare il salto relativo e poi quello in sequenza. **Tuttavia uno stesso byte può essere associato ad istruzioni macchina differenti**. Il disassembler non può risolvere ciò, perchè mentre **per lui non è possibile quella doppia associazione, per il processore è possibile**. Ne è un esempio `eb ff c0 48`, i primi due byte corrispondono a `jmp -1`, è un salto relativo in cui prendiamo instruction pointer, ovvero `c0` (perchè l'IP punta all'istruzione seguente), sottranedo `-1` torniamo al codice operativo `ff`, ovvero associato a due byte `ff c0` cioè `INC EAX`, mentre `48` è `dec eax`. Quindi per `ff` abbiamo associato sia `jmp -1` sia `inc eax`, e il disassembler non riesce a risolvere questo contenzioso. Il problema è che `ff` ha quindi due ruoli: offset del salto e codice operativo dell'istruzione da eseguire.~~~~

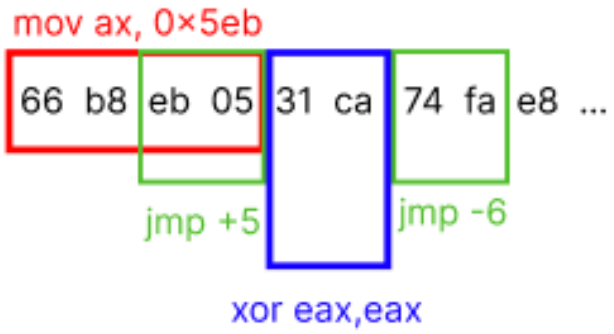



Il risultato "finale" modifica i flag di stato (se coinvolti!).

**AntiDis1.exe**

Apriamo debugger su Windows, ovvero *OllyDbg*. Mettiamo breakpoint su `ad157b`, mettiamo breakpoint e otteniamo un crash. Il discorso è che un salto relativo allo stesso segmento può falsare ciò che il disassembler ci fa vedere. Il disassembler vedeva `jmp -1` e separatamente ciò che viene dopo `c0`. (Il riferimento è sempre la foto vista prima!)

Per il disassembler, **un byte appartiene ad una sola istruzione**, quindi vede quella giusta o quella sbagliata. Nessun compilatore genera un salto a qualcosa di già eseguito, quindi è un pattern scovabile solamente nei malware. Prendiamo `66 b8 eb 05 31 ca 74 fa e8`




Si parte dai primi 4 byte, ovvero `66 b8 eb 05`, cioè `mov ax, 0x5eb`. Il sottoinsieme composto da `eb 05` corrisponde a `jmp +5`. Gli altri 2 byte `31 ca` corrispondono a `xor eax,eax` ed infine `74 fa` è l'opcode di `jmp -6`, che ci fa tornare indietro ad `eb` e prende `eb 05`, quindi salta in avanti di +5 (partendo dal byte `31`), saltando dopo `e8`. In **antiDis1.exe**, al riferimento `401553` si ha `byte 66 b8 eb 05 31`, e prendiamo le tre istruzioni fino al `jz`, vediamo che c'è una lable ma poi la call è in **rosso**. Ghidra ignora il salto, il codice è valido ma non vede entrambi i rami. Questo perchè un byte è di una sola istruzione macchina. Se vogliamo vedere il secodo ramo, togliamo l'interpretazione, (ci troviamo al riferimento `00401555` che non mostra criticità). Saltando il byte `e8` risolviamo il conflitto.

## Funzioni dipendenti da parametri

**Antidis2.exe**

Tutti i programmi ad oggetti sono difficili da dissasemblare, in quanto il programma non contiene indirizzi esatti delle funzioni, ma indirizzi costruiti tramite tabelle, perchè i metodi virtuali dipendono dall'*istanza dell'oggetto*, facendo call a registri caricati precedentemente che il disassembler non conosce. Possiamo farlo anche nella programmazione da noi trattata, compiendo operazioni inutili ma di intralcio all'analista. Al riferimento `00402931` vengono messi due valori dentro lo stack, seguiti da dei calcoli e conclusi da una `call eax`, al riferimento `0040295c`. Vediamo che la **funzione dipende dal parametro fornito**, quindi se facessimo il grafico delle funzioni, noteremmo che queste call sono **oscurate**. E' bastato prendere i valori veri di salto, poi delle operazioni (come trasformare il valore in base all'input). Al lancio del programma, esso funziona su Windows, perchè passiamo 0 argomenti, e nel corrispettivo codice shiftiamo di 0 posizioni, quindi non abbiamo problemi. Se lanciassimo il programma con qualche argomento andremmo in crash. Per il malware va bene, perchè è un ostacolo per l'analista. A livello di programma in C, ciò che viene fatto è prendere pointer, usarlo per puntare ad una funzione, inizializzarlo e manipolarlo. Comedi seguito:

```
void (*volatile p)(void) = crash_debugger;
void (*volatile q)(void) = show_messagebox;
p = (void(*)(void))(long)p>>(argc-1);
q = (void(*)(void))(long)q>>(argc-1);
p();
q();
return 0;
```

## Structured Exception Handlers, SEH: falsare il flusso di esecuzione

**AntiDis3.exe**

Possiamo falsare il flusso di esecuzione. Vediamo il seguente codice:

```
    call l0
l0: add [esp], 5   // 4 byte
    retn           // 1 byte
l0+5: ...
```

Quando eseguiamo la `call l0`, mettiamo nell'indirizzo ritorno di `l0` **se stesso**. Ovvero `esp` punta ad `l0`. (In quanto, al return della `call`, si parte dall'istruzione successiva alla `call`, ovvero la label `l0`). Quindi `esp` punta ad `l0`, l'indirizzo di ritorno. Poi sommiamo 5, e quindi sullo stack abbiamo `l0+5`, ovvero l'**indirizzo dell'istruzione seguente ad *add***, in quanto l'istruzione di `add` è lunga 4 byte! Con la return, prendiamo dallo stack tale indirizzo dalla cima dello stack e ci salta dentro!

Ciò vuol dire che le istruzioni da `call l0` a `retn` è come se non ci fossero state, come una grande `NOP`. Dallo stack viene tolto `l0+5`, e si riparte da `l0+5`.

Per Ghidra la funzione finisce alla return, noi manipoliamo ciò che c'è sullo stack per alterarne il comportamento. La sequenza di istruzioni ha portato il processore in maniera complicata a continuare l'esecuzione e, come se fosse una grande `NOP`, dal punto di vista del disassemblatore è cambiato tutto, perchè sto modificando il flusso di esecuzione. Il disassemblatore pensa che la funzione finisca dove incontra una return. Ghidra non riesce a riconoscere il fatto che ci siano delle altre `CALL`.

## Sfruttare la divisione per 0 - eccezione SEH

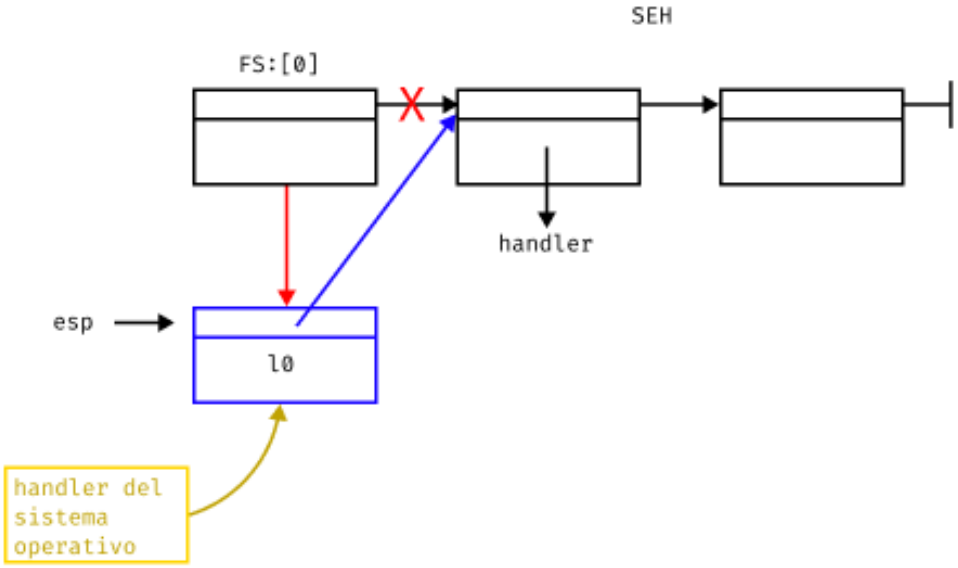
```
    mov eax, 0x14              //metto 20 in eax
    add eax,(offset l0)-0x14   //  offset tra l0 e 14,
    push eax                   // metto nello stack tale indirizzo
    push dword ptr fs:[0]      // Puntatore a primo elemento di catena di handler
    mov fs:[0], esp            // metto esp in FS[0], cioè FS[0] punta a stack
    xor ecx, ecx
    div ecx                    // divisione per 0, genera eccezione
    ret                        // return mai eseguita
    ...                        // altre istruzioni non eseguite!

l0:
```

```
mov esp, [esp+8]          // recupero stack pointer, +8 perchè il SO ha un HANDLER AGGIUNTIVO
mov eax, fs:[0]           // leggo in eax il contenuto di fs:[0]
mov eax,[eax]             // dereferenzio eax (ovvero punto al primo handler)
mov eax,[eax]             // dereferenzio di nuovo (punto all'ultimo handler)
mov fs:[0],eax            // in fs:[0] ci metto eax, cosi ci punta direttamente
add esp,8
```

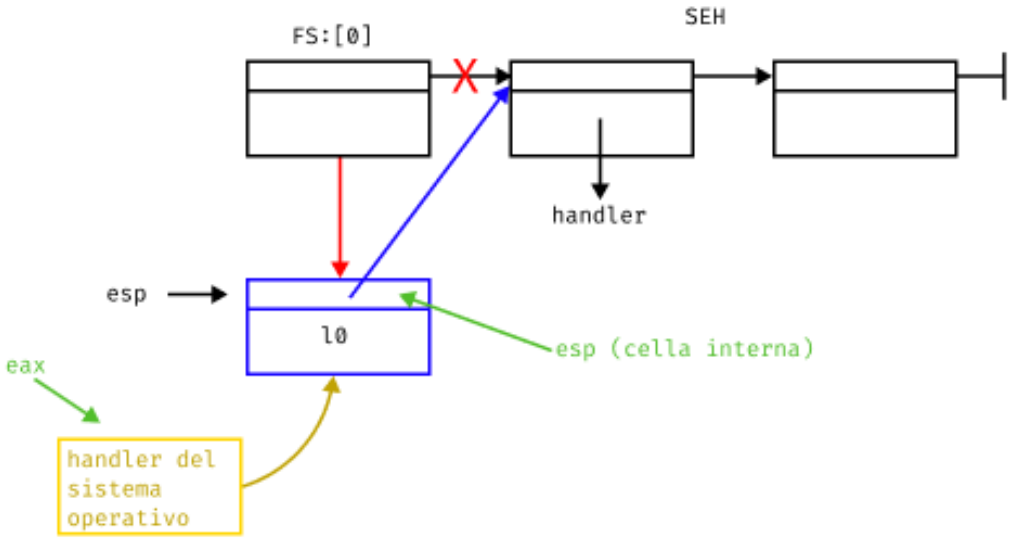
Inizialmente, il programma modifica la catena degli handler delle eccezioni (SEH) inserendo un nuovo handler.

1) In blu vediamo che è stata fatta la push di `l0`, e poi la push di `FS:[0]`, che punta al primo handler. Questo nuovo handler viene inizialmente puntato da `FS:[0]`. Poi `FS:[0]` punterà ad `esp`:

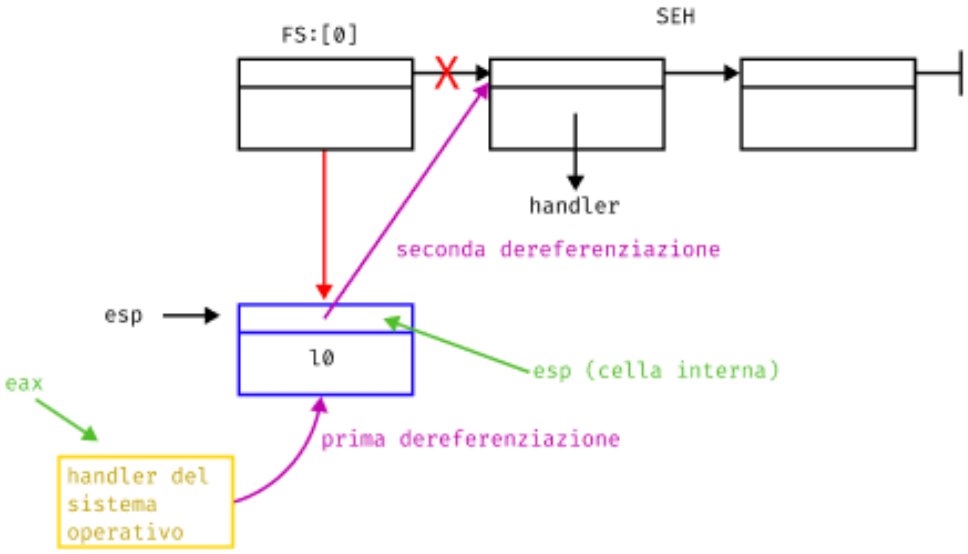



L'eccezione generata attiva il gestore delle eccezioni. Tuttavia, questo gestore è progettato per non restituire il controllo al programma principale. Ovvero non passiamo mai per `return`.

2) Passando al `l0`, viene recuperato lo stack pointer e messo nella cella di `esp`, inoltre `eax` punta all'*handler del sistema operativo*.




3) Dereferenziamo due volte `eax`:




Rimuovendo infine il box blu, aggiungendo 8 ad `esp` , cioè togliamo dallo stack le cose introdotte, in quanto non ci serve più.

Da un punto di vista del SO, questi è convinto che stiamo ancora eseguendo un gestore delle eccezioni, ma all'atto pratico il programma funziona. Alla fine sono dei salti! Per ricostruire il flusso si fa clear di tutto ciò che non serve e si mette un jump dove si vuole andare.

#### AntiDis4.exe

`Fun_00401530`, Ghidra capisce che a riga `0040153d` punta poi a `LAB_00401560`, anche se non ci sono collegamenti logici.

#### AntiDis5.exe

```
    sub esp,4
    cmp esp, 1000h
    jl l0
    add esp,4
    jmp l1
l0:
    add esp, 0x104
l1:
    access stack
```

Il disassemblatore non ha modo di sapere se `esp` è quello originale, dato solo dalla prima istruzione di `sub`, oppure se ci sono anche le due `add` che variano lo stack pointer. L'effetto pratico è che aggiungendo valori allo stack pointer si prendono valori di altre funzioni, quindi valori di funzioni chiamanti. Tale approccio quindi non può funzionare, ma questo il disassemblatore non lo può sapere, portando ad una errata traduzione delle variabili locali. Ghidra tiene traccia di ciò che c'è sullo stack per una relativa funzione. Nel codice, il confronto `cmp` è sempre falso, perchè **lo stack non sta mai nella prima pagina, bensi in indirizzi molto più alti**. Se invece il disassemblatore decide di prendere la strada sbagliata, salta la logica della ricostruzione delle variabili locali.

### Misure anti- Virtual Machines

Simili a quelle anti debugger, ovvero il malware non vuole eseguire su MV perchè capisce di lavorare in una sandbox di un'analista.

#### CPUID - AntiVM1.exe

La misura di protezione di questo programma non ha effetto, e quindi funziona tranquillamente. Cosa non ha funzionato? C'è il comando assembly `cpuid`, che su Intel ritorna stringa `GenuineIntel` e su AMD `AuthenticAMD` se non passo

alcun argomento su `eax`. Su VmWare, la stringa ricevuta è differente, e ritorna una stringa relativa a `processori VMWare`. Qemu, in questo caso, non genera tale problema. Tale malware controllava proprio questo.

#### AntiVM2.exe

Altro caso: `cpuid` con `eax = 1`. Ritorna una quantità elevata di flag associati alla VM, ad esempio, nel registro `ecx`, il 31-esimo bit è libero, e spesso viene usato per segnalare se si lavora su VM ( = 1) o meno (= 0). Fa proprio questa cosa.

**Come porre rimedio?** Mettiamo `NOP` su questi check o comunque patchare l'eseguibile, altra alternativa: modifichiamo qualche settings delle VM.

#### AntiVM3.exe

Anche lui usa `cpuid`, sfruttando codice `0x40000000`, che ritorna chi ha costruito la macchina virtuale ("vboxvboxvbox", oppure "vmwarevmware" o "kvmkvmkvm"...). `ecx`, `ebx`, `edx` sono i tre registri usati, basta vederne uno.

### Vedere nelle chiavi di registro (regedit)

Cerchiamo tutto ciò che è associato all'installazione di una macchina virtuale. Ad esempio, Vbox, per avere cartella condivisa, sicuramente installa qualcosa, e lascia tracce nei registri di sistema. Possiamo quindi trovare stringhe associate a queste macchine virtuali.

### Controllare i processi in esecuzione

Il comando *superuser* `wmic process | fomdstr /i vbox` cerca processi in esecuzione. Per questo è bene non installare cose che possano essere associate alle vm. Lo stesso ragionamento viene fatto per i **servizi**, usando `net stat | findstr /i vmware`, ovvero cerchiamo servizi contenenti la stringa "vmware". Ancora, possiamo trovare i **file** o i **driver** associati ad una macchina virtuale. Qemu non usa estensioni proprie per realizzare cartella condivisa, bensì usa *samba*, non lasciando le stesse tracce di altre virtual machine.

### Istruzioni vulnearbili

Altre tecniche che possono essere utilizzate si basano su informazioni meno note (meno documentate).

Ci sono istruzioni macchina particolari:

- **SIDT**: interrupt descriptor table
- **SGDT**: global descriptor table
- **SLDT**: local descriptor table
- **CPUID STR**: store register
- **SMSW**: scrive in un registro o in una locazione di memoria il dato a 16 bit contenuto nella Parola di Stato della Macchina (`Machine Status Word`)

Servono al sistema operativo per manipolare strutture dati che servono al processore. Possono essere eseguite da processi normali per restituire posizioni di tabelle che servono al processore. Sono gestite dal SO e il processore deve sempre poter accedere a queste tabelle. Non possono modificare il contenuto o i puntatori a queste tabelle ma possono farle vedere. Molte VM modificano il formato delle tabelle puntate da queste istruzioni, il codice può accorgersi se esegue o no in una VM.

### Tecnica pill

Famiglia di tecniche usate da un malware per capire se la sua esecuzione è realizzata su una macchina fisica o su una macchina virtuale.

#### Red Pill

Funzionava con i processori con un solo core ora non più. Basata su `sidt`.

#### No Pill

Basata su `SGDT` e `SLDT`. Vede a cosa puntano e il loro formato, per capire se si è dentro una VM. Si può configurare la VM perché le cose continuino a funzionare bene e quindi il malware venga ingannato, oppure patchare il malware. `SMSW` salva i registri di stato, ma il formato cambia leggermente se si è dentro una macchina virtuale o meno.

Ultima categoria: ci sono macchine virtuali che implementano processori virtuali che hanno cose in più di quelli normali.

### Processori virtuali e servizi addizionali

In VMWare posso usare:

```
mov eax, `vmxh`
mov ebx, address
mov ecx, 0ah
mov dx, `vx`
in eax,dx
```

Una macchina virtuale può usare queste istruzioni per *creare un canale di collegamento tra sistema guest e macchina virtuale.* Il malware prova ad usare questo sistema, se qualcuno gli risponde è la VM (in questo modo capisce che sta girando su una VM). Se uso servizi come le *Guest Additions*, il codice può accorgersi delle tracce lasciate da queste (ad esempio dai valori dei registri) e quindi del fatto che sta eseguendo su una VM. `IN` e `OUT` sono supportate dai vecchi sistemi opertivi, ma **non da quelli moderni**, perché riferenti a indirizzi differenti da quelli normali. Oggi sono *memory mapped*, ovvero indirizzi particolari. Se mmappato, si arriva a registro hardware, prima si usava IN e OUT. Ad esempio, *Vmware* le usa per collegare guest e vm. Anche l'uso delle guest addiction può portare alla generazione delle tracce. Sarebbe meglio non usarle, ma ciò non ci dà certezza dell'assenza di altre tracce presenti.

## 12 dicembre 2023

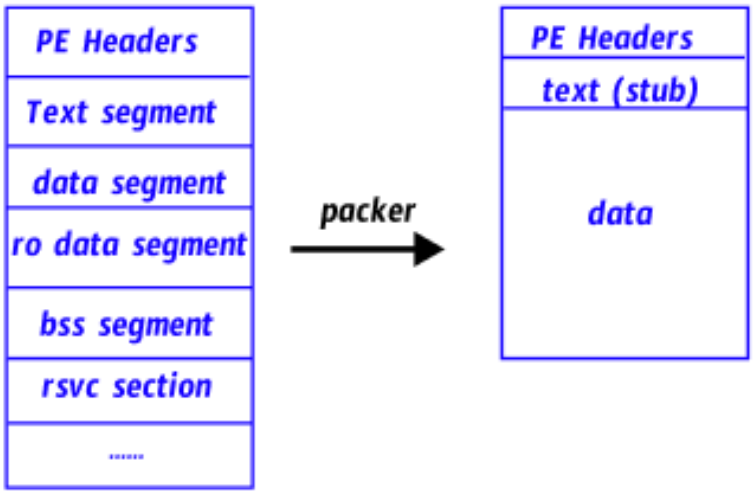
### Programmi protetti da packer

I packer nascono con sistemi aventi poco spazio disco. C'era necessità di ottimizzare lo spazio. Si volle quindi comprimere la dimensone sul disco. Oggi la tecnologia è rimasta, ma non per motivi di memoria, bensì come meccanismo di difesa sfruttato dai malware.

#### Come è fatto un eseguibile

Il compilatore produce eseguibile, avente struttura consolidata. Partiamo da delle *testate PE*, poi segmenti *testo*, *dati*, eventualmente in Windows anche *risorse* etc. . . , come in figura:


Quando applichiamo ad un file **un packer**, l'eseguibile ottenuto fa esattamente le stesse cose dell'originale, *ma la struttura è totalmente diversa.* La testata sarà diversa, la sezione testo viene detta *stub*, e un'unica sezione *data.* Tutte le sezioni appena discusse confluiscono mediante un algoritmo di compressione dati, con un'immagine più piccola sul disco.




Lo **stub** legge i dati in sezione *data* per ricostruire in memoria l'immagine che si avrebbe come se si fosse eseguito il programma originale. **Lo stub non ha nulla a che fare col programma originale, capire lui non è indice di aver capito il programma originale.** Il programma è compresso sul disco, e viene decompresso in memoria. Esistono algoritmi di **scrambling** per ostacolare l'analista, in particolare lo stub viene complicato con istruzioni volte a rendere più difficoltosa l'analisi.

In linea di principio, l'analista può sempre ricondursi alla versione originale, tuttavia con queste tecniche si punta a rallentare le tempistiche o a renderle antieconomiche. Il sistema operativo carica sempre il programma *packed*, che non usa niente che verrebbe usato dal programma originale. E' compito dello *stub* mettere a posto le cose.

#### Operazioni dello stub

1) Esegue l'operazione di **unpack** dell'eseguibile originale, mettendolo in memoria RAM.
2) Deve risolvere gli **imports**, ovvero i *record* che ci dicono cosa viene usato di una certa libreria, quindi quale API di quale `.dll` viene usata.
3) Si salta all'*entry point* originale. Il nostro file eseguibile originale aveva un certo **Original Entry Point**, espresso nella testata. Quando viene decompresso, il file deve partire da questo punto.

La cosa più delicata è capire la risoluzione degli *imports*, ovvero il punto 2.

Noi vorremmo, su disco, un file che si comporti e funzioni come il file originale. Cioè vogliamo disfare il lavoro dell'*unpacker*, quantomeno per lavorare col debugger.

Tipicamente, lo stub lavora all'inizio, fa il salto all'**OEP** e termina il suo ruolo.

**NB**: Packer più complessi potrebbero scindere una sezione in più blocchi, portando lo stub ad eseguire la decompressione di ogni blocco quando necessario, quindi in memoria non ci sarà mai il file *completo*.

## Fasi del packer

#### Fase 1, operazione di packing

- Non c'è molto da dire, ogni unpacking può usare un proprio algoritmo di *compressione*.
- Può usare algoritmi di *scrambling*, ovvero i dati compressi possono subire variazioni. Se l'algoritmo è troppo banale, non c'è bisogno dello stub.

E' possibile creare un *unpacker*, cioè applicare ingegneria reversa ad un *packer* per capire come funziona. Esiste anche la possibilità di creare *plugins*, che spesso non vengono *venduti* insieme ad un *packer commerciale*, poichè aggiungendoli alla fine, si evita la possibilità di vendere un programma completo e completamente analizzabile.

La fase di unpacking è lasciata allo stub.

**Fase 2, gestione degli imports**

Esiste **Import Address Table**, che ci dice quali *API* di quali `.dll` vengono usate. Questa tabella è sempre gestita dal *sistema operativo*.
E' compito dello stub *sistemare* questa tabella *IAT*, in tre modi:

- Viene mantenuta la tabella originale *IAT* nel programma impacchettato. Quindi, come avevamo una *IAT* nel file originale, la avremo *uguale* anche nella versione compressa.
  - *Vantaggio*: è facile, la risoluzione viene eseguita dal sistema operativo. Il fatto che lo stub non le usi, non è un problema, perchè se ne preoccupa il Sistema operativo quando vede la *IAT*.
  - *Svantaggio*: Questa tecnica va bene se non siamo interessati a proteggere le informazioni, ma solo per la compressione, in quanto leggendola, ci fornisce informazioni su cosa viene usato.
- Viene mantenuta la tabella originale *IAT* ma, per ogni libreria dinamica `.dll`, viene lasciata **solo un'API**, rimuovendo tutte le altre. Ad esempio, se abbiamo *user32.dll* che sfrutta *fun1, fun2, fun3* e *kernel32.dll* che sfrutta *fun4, fun5*, verranno lasciate solo *fun1* di *user32.dll* e *fun4* di *kernel32.dll*. Questo perchè le varie *API* sono in un **ordine prestabilito**, quindi, nota la versione della libreria, e posizione della prima funzione, possiamo ottenere anche le altre. Lo stub sa cosa è stato rimosso, quindi sa che deve ricostruire *fun2,. . . , fun5*.
  Nell'eseguibile, in memoria, c'è sempre la tabella compressa. Inoltre, quando in Ghidra vediamo il richiamo ad una *.dll*, verrà eseguito un *jmp* ad essa, in quanto a runtime la posizione cambierà. E' grazie alla IAT che sappiamo dov'è la collocazione di questi *wrapper intermedi*.
- Non usiamo il sistema operativo, quindi non si copia la *IAT*, tutti i simboli *vengono sistemati dallo stub*. Vengono sfruttate le seguenti *API*:
  - `LoadLibrary`: carica la *.dll* nello spazio del processo.
  - `GetProcAddress`: passiamo il nome o il suo numero dell'API, ottenendo l'indirizzo.

  Così lo stub carica le *.dll* che servono, ottiene l'indirizzo giusto e lo scrive nei wrapper. Il programma originale farà quindi una chiamata al *wrapper*, che indirizzerà l'esecuzione verso queste librerie.
- Non copiamo la *IAT* nè usiamo le due api esposte precedentemente. Per ora non sarà trattato.

**Fase 3, salto all'original entry point.**

Sapere il salto all'*original entry point* ci permetterebbe di passare alla fase di analisi, in quanto tutto il codice precedente sarebbe *stub* di non interesse. La difficoltà risiede nel *capire dove questo salto all'original entry point viene eseguito*.

Tuttavia, non è detto che lo stub faccia semplicemente un *jmp*, in quanto lo stub potrebbe offuscare questa tecnica, magari *posticipando* il salto.

Per ora, vediamo il caso "*semplice*".

## Programma upx

**orig.exe**

Vogliamo impacchettarlo (che nel periodo Natalizio è cosa buona e giusta). Useremo **UPX**, programma semplice ed open source. Da linea di comando:

- Per comprimere: `upx -o filepacked.exe orig.exe`
- Per decomprimere: `upx -d -o unpacked.exe packed.exe`

I file prodotti, a livello d'esperienza uso, sono uguali.

**Riconoscere un file packed**

Un modo è leggere il programma con un *tool* di analisi statica, ad esempio con *PEStudio*. In questo modo, possiamo vedere le librerie, le stringhe ed i nomi delle API per la risoluzione della *IAT* (come `MessageBox`). Anche vedere i nomi delle sezioni è utile: in questo programma troviamo tutte le sezioni a noi note.

Facendo la stessa analisi, con la versione *packed*, vediamo che:

- Le sezioni hanno nomi diverse: si chiamano `upx0` (lo *stub*), `upx1` e `upx2` (che contengono la sezione *data*) , infine`resources`. **Questo perchè *upx* non nasconde ciò che fa.**

- Nelle *API* usate troviamo solo:
  - `virtualProtect`: cambia i diritti di accesso alle pagine di memoria, lo stub deve usarlo. Intel è *eseguibile o scrivibile*, quindi bisogna cambiare i diritti di accesso.
  - `LoadLibrary` e `getProcAddress`: visti in precedenza, per capire le *.dll*
  - `messageBoxA`: messaggio che il packer mostra se il file è corrotto, ma non è la message box originale.

Aprendo il programma packed con **PeBear**, si vede una netta **differenza tra *virtual size* e *raw size*, indice di compattazione del file**, in quanto in memoria occuperò più spazio (o meno spazio) rispetto a quando sono su disco.

Possiamo usare uno strumento di analisi dinamica (che esegue il codice) come **PEid**, programma semplice avente plugin *generic unpacker*. Passando il programma impacchettato, ci dice anche la versione usata di *upx*. E' possibile anche *provare* a trovare l'original entry point, però molto spesso fallisce. Quantomeno ci fornisce l'informazione sul fatto che sia stato impacchettato.

Altro metodo è usare il buon vecchio *Ghidra*, dove ci aspettiamo di vedere molti `DAT` (cose che non riconosce), e non tante istruzioni.

Anche le stringhe sono analizzabili, ma non è un metodo sicurissimo, in quanto non sappiamo quali siano le stringhe originali.

Altra tecnica è misurare la quantità di informazione nelle sequenze di byte, ovvero stiamo parlando di *entropia*.

### Entropia

Prendiamo un blocco di 64 volte 0, dove ogni 0 rappresenta 00000000. Letto un byte, gli altri sono tutti uguali, quindi l'entropia è **minima**, ovvero 0.

Se generassi numeri random, l'entropia sarebbe **massima**, il valore in posizione $i$ non mi aiuta a prevedere il valore di $i+1$. In ogni byte, tutti gli 8 bit sono significativi, in quanto non prevedibili, questo è il valore massimo.

I byte non sono realmente casuali, quindi l'entropia non è mai massima, bensì compresa tra 5 e 6, quindi 2/3 byte non sono significativi.

La **compressione alza l'entropia**, in quanto vengono tolte cose ridondanti, alzando l'entropia a valori tra 7/8.

In **PeStudio** esiste il campo **entropy** per ogni sezione, e nel file compresso si attesta su 7.
Se uno volesse l'entropia totale dovrebbe usare il fantastico programma scritto dal professore, ma non fornito.

In blocco di 64 byte, l'entropia si attesta su 6, in quanto $2^6 = 64$.

## 14 dicembre 2023

Abbiamo visto le azioni compiute dallo stub:

1. unpack
2. risoluzione IAT
3. jump to OEP, detta anche *tail jump*.

Uno strumento per spacchettare, oltre a *upx*, è **PEXplorer** (licenza a tempo), il quale dispone di tre motori di decompressione (upx ed altri due). Se non riuscissimo ad usare *tool automatici*, possiamo percorrere due strade:

- analisi lunga e faticosa sullo stub, facendone poi il reversing. Stiamo però perdendo tempo su qualcosa che non ci aiuta a capire cosa farà veramente il programma in sè.
- approccio semi-automatico, che andiamo ad analizzare.

### Approccio semi-automatico per il reversing dello stub

1. La prima cosa che dobbiamo fare è capire quale sia l'**OEP**, mediante analisi statica o dinamica, rimuovendo eventuali meccanismi anti-debugger.

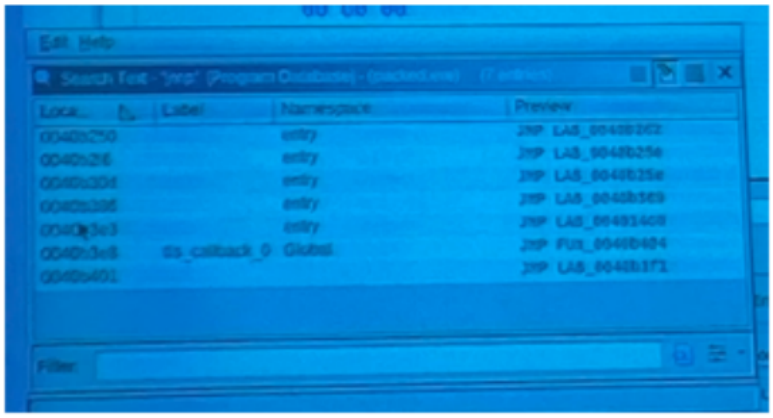
2. Mettiamo un breakpoint al salto all'OEP, per capire quando lo stub finisce di lavorare. Facciamo poi un **dump**, una copia su file di tutta l'immagine costruita dallo stub. Questo dump viene messo su un file *non eseguibile. Quando lo stub opera, lo fa in memoria, sta sistemando gli indirizzi del wrapper, non correggendo la IAT* (magari non gli serve). **Il dump è simile alla struttura di un file eseguibile (è una sua compressione) ma mancano/sono assenti le info sulla *IAT*.**
3. Importiamo la **IAT**. Questo è lo step più complesso, dipende da come lo stub ha risolto i riferimenti. Tuttavia, spesso, basta risolvere i primi due e lavorare con il *disassembler*, mettendo da parte il debugger.

## 1 - Localizzazione dell'OEP

### Ricerca delle long jump

Prendiamo Ghidra, e il nostro programma `packed` compresso. Ghidra ci capisce poco, però possiamo cercare nelle *instruction memories* la parola `jmp`, trovando le occorrenze.

Lo stub è piccolino, con molti cicli, in cui lavora sulla sezione *data* nello spazio di indirizzi che il SO ha usato per il programma impacchettato (grande dimensione) e su pagine di memoria allocate dinamicamente sull'*heap*. Qui troviamo l'*OEP*. Allora dallo stub (nello spazio di indirizzi) farà un salto *lontano* all'OEP nell'*heap*. Quindi dobbiamo vedere la distanza del salto.



Da `4014c0` a `01` c'è una differenza di diverse decine. Tuttavia, saltando, troviamo svariati "`??`", perchè non siamo nell'heap, ma nello spazio di indirizzamento.

### Tool automatici

Possiamo dare il tutto in pasto a **PEid**, che però non ci dà aiuto in questo caso.

**Attenzione**
Ricordiamo che una `push 0x4014c0`, seguita da una `return`, è equivalente all'istruzione `jmp 0x4014c0`.

### Breakpoint sullo stack

Altra strategia, invece di usare il disassembler, è usare il debugger (a patto che non ci siano meccanismi anti-debugger). Questo è utile se troviamo istruzioni `PUSHAD`, in quanto *il packer non sa cosa farà il programma che sta impacchettando*, e quindi appena parte si **salva tutti i registri sullo stack**. Quando il programma originale parte, gli fornisco i valori nei registri come configurati dal sistema operativo.

Passando al debugger, eseguiamo questa `PUSHAD`. Mettiamo un breakpoint hardware (poichè trattiamo un accesso sullo stack) sul registro che è stato salvato sullo stack (probabilmente ci sarà la POP che rimetterà a posto i valori in quei registri). Si ferma dopo una `POPAD`. Allora ci aspettiamo una **tail jump** nelle vicinanze. Ovvero, mi aspetto di fare un jump al programma originale e solo dopo ripulire lo stack con POP. Troviamo un loop che mette alcuni 0 sullo stack, lo saltiamo e troviamo finalmente la jump. Ora vedo codice disassemblato che prima non vedevo con Ghidra. Ricordiamo che è scomodo esaminare il disassemblato da qui, in quanto il nostro obiettivo è ricostruire l'eseguibile.

E' buona norma mettere dei **breakpoint on loop** sullo stub, skippandoli fino a che non arrivo all'entry point.

## 2 - Dump della memoria

Qui usiamo **PEid**, facendo l'unpacking e fornendo l'entry point che abbiamo trovato nello step precedente. La dimensione sarà più grande del file originale, perchè non sa precisamente quali pagine di memoria sono accedute, e quindi *ne prende di più del necessario.*

Alcuni plugin utili in OllyDbg sono:

- Olly advanced
- bookmarks
- command line
- *ollydump*: offre modalità *trace into* (passo dentro le call) e *trace over* (ci passa sopra). Tipicamente solo uno delle due funziona. Dobbiamo togliere breakpoint hardware eventualmente messi. Se non vedo nulla, probabilmente si è perso come Pollicino. Tipicamente proverà a fare i passi 2 e 3 insieme (identificato dall'opzione *rebuild import*).

Possiamo mettere all'OEP un hardware breakpoint di tipo `execute`, se funziona mi porta sulla riga a cui eravamo interessati, riproviamo a fare il dump da qui. Facciamo sia con *rebuild import* che *senza* (proviamo tutto). Senza *rebuild import* l'eseguibile non partirà mai (non ha *IAT*), con *rebuild import* l'ha prodotta ma probabilmente male. Per vedere se il *dump* è corretto, basta eseguirlo (a nostro rischio e pericolo).

Proviamo con altro software: **xDBG** (32 o 64) che dispone del plugin *Shilla* per il dump.

## 3 - Import Reconstruction IAT

Esiste tool **ImportREC**, tool così potente che, senza disattivare *Windows Defender*, viene automaticamente rimosso. Si esegue con diritti di amministratore. Senza *OEP* non funziona.

Dobbiamo fornirgli:

- Processo attivo, in cui riconosce le `.dll` caricate.
- In basso a sinistra, richiede l'OEP (l'offset dell'OEP), quello proposto è sbagliato, in quanto è quello dello stub, e poi *auto-search* (quello che abbiamo trovato è `4014c0`). Per capire se ciò che forniamo è corretto, facciamo `getImports`. Possiamo fare `showInvalid` o `showSuspect` (non è convinto del range di indirizzi). Se la maggior parte sono *sane*, probabilmente è quello corretto. Concludiamo con `fixDump`, fornendogli il file dump (sia con *IAT* o senza, è uguale; lui la aggiunge o aggiorna!). Verrà creato un nuovo file contenente alla fine il carattere `_`, che ora funziona se avviato! Nulla vieta di fare questa cosa **ricorsivamente**. (*sicuramente il malware finale sarà fatto così*, affermazione susseguita da risata malefica)

**NB**: Se non funzionasse *ImportREC*, `x32dbg` o `x64dbg` con plugin *Shilla* fornisce una valida alternativa.

## Esercizio lab18-01.exe

1. Analizziamo con **PEstudio**, ci dà qualche info sulle imports, sezioni *text*, *data*, *upx2*. Quindi è compresso.
2. Analizziamo con **PEid**, che ci dice che è compresso con `upx 0.49.6`
3. Proviamo a scompattarlo con **Upx**, `upx -d lab18_01.exe`, ma il risultato è un fallimento. Non ci riesce.
4. Proviamo con **PExplorer**, il volume non è aumentato, non ha fatto nulla.
5. Allora faccio *a mano*. Apriamo con **Ghidra**, `search -> program text`, cerchiamo *far jump*. L'ultimo jump visto pare lunghino. Non vediamo nulla se saltiamo, è promettente. Riferimento `409f43`
6. Andiamo su **OllyDbg**, carichiamo il programma, ci informa che il file è compresso. `CTRL+G` e saltiamo a `409f43`, mettiamo `breakpoint`, eseguiamo fino a qui, e ci entriamo con `F7`, e vediamo cose note: *exitProcess*, *getVersion*,...
7. Facciamo **OllyDump**, mettendo il riferimento, e testiamo sia con che senza *IAT*. Funziona? non sappiamo cosa fa, quindi chissà. Non dà problemi, non si è rotto nulla (pare), forse sta funzionando.
8. Proviamo con **ImpREC**, lanciamo il programma *packed* (quindi quello iniziale) così rimane in memoria. Lo selezioniamo, e specifichiamo il riferimento `154f`, trova qualcosa, trova tre imports, facciamo `fixDump`, salviamo il nuovo dump. Non è cambiato nulla, ma probabilmente è proprio il programma che non fa nulla.

# 19 dicembre 2023

## Meccanismi sfruttati dal malware

Il malware viene scritto per un certo motivo, e può operare con diverse modalità. Classifichiamole, ricordando che una non esclude l'altra:

- **Persistenza**: dopo la prima infezione, il malware vuole continuare ad avere il controllo della macchina.
- **Privilegi**: il malware viene spesso lanciato da utente normale, tuttavia richiede privilegi superiori.
- **Segretezza**: il malware vuole nascondere l'esecuzione all'utente. Ciò varia dal tipo di malware, ad esempio il randsomware è piuttosto evidente rispetto ad altre tipologie. Tuttavia, all'inizio dell'esecuzione, tutti vogliono rimanere nascosti.
- **Evasione**: Esistono Firewall per verificare se il software presenti comportamenti anormali, e per bloccarli vengono definite specifiche regole (*ad esempio*: blocca connessione sulla porta 80 ad un server remoto, se non si utilizza Edge.)

## Persistenza

Il malware tenta di riavviarsi insieme al computer. Spesso sono coinvolte le chiavi di registro, usate da Windows per memorizzare le configurazioni. Il meccanismo più comune sfrutta la chiave:
`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`, contentente le chiavi delle applicazioni che vengono lanciate. Ciò è visibile con `regedit`. Non è l'unico posto in cui vediamo i programmi avviati allo startup, quindi se qui non troviamo nulla, non vuol dire che non ci sia il malware.

Esistono tool di sistema **SysInternals**, tra cui c'è **Autoruns**, per gestire tutti i programmi che partono all'avvio.

### Uso delle dll

Ovviamente, esistono altri meccanismi per la persistenza. Vediamo **AppInit-dll**: I programmi *.exe*, utilizzanti GUI, sfruttano `user32.dll`, contenente API molto comuni. Esiste un meccanismo ufficiale che aggiunge altre `.dll` oltre a questa, bypassando tutti i criteri di sicurezza.

Pensiamo di creare `malware.dll`, anch'esso è file eseguibile. Il `file.exe` ha un solo entry point, mentre una `.dll` ha più entry point, *uno per ogni API*. Esiste una `dllMain` (nome generico, è l'indirizzo presente nalla Testata del PE header), usato quando la `.dll` viene collegata ad un processo, quindi esegue azioni di inizializzazione. Appena viene linkata, il processo va ad eseguire anche il codice di `malware.dll`.

Se colleghiamo questo discorso al fatto che, mediante `user32.dll`, è possibile collegarci a `malware.dll`, capiamo la gravità della situazione.
In `HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows` troviamo tutte le `.dll` iniettate con `user32.dll` (quindi non sono *interne* ad `user32.dll`, è questo il problema!) In questo percorso troviamo `AppInit_DLLs`, inizialmente vuota e sfruttabile per queste operazioni. Ovvero, tramite questa `.dll`, vengono caricate tutte le dll (anche malevoli) che fanno uso di `user32.dll`.

### Winlogon Notify

Sono azioni legate al *logon*, *logoff*, *startup*, *shutdown* o *lock screen*.
A queste attività viene associata automaticamente una `.dll`, che troviamo a:
`HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon\Notify`, che nell'esempio del prof non è presente. Basta un'attivazione unica per rimanere persistente.

### Windows Services

Vogliamo aggiungere la segretezza, lo si fa nascondendosi tra i servizi di Windows (come i demoni in Linux/Unix), visibili con `services.msc`. Questi programmi *potenzialmente partono allo startup* (dipende dalla configurazione). Il modo più semplice (non il più segreto!) è fornire al malware un nome fittizio ed anonimo. I servizi richiedono la manipolazione dei servizi di sistema, visibili a: `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\ServiceName`

**Attenzione**: `CurrentControlSet` non è il vero contenitore di questi settaggi, bensì è un puntatore al controlSet vero! Per questo, è possibile trovare `ControlSet001` ed altri! Quando creiamo un **punto di ripristino**, viene mantenuta la vecchia configurazione, ovvero il `CurrentControlSet` punta alla versione precedente.

Il malware aggiunge il servizio a tutti i `ControlSet`, non solo su quello attuale. Scendendo nella sottocartella `CLFS` troviamo un `ImagePath`, percorso del servizio (in questo caso eseguibile del malware). Troviamo ancche `Start`, che ci dice come parte il servizio. Se = 2, allora il servizio parte *automaticamente*.

Non tutti i servizi hanno un proprio eseguibile per essere implementati, e quindi si appoggiano ad un processo contenitore, ufficiale Microsoft, chiamato `SVCHOST.exe`. La lista dei processi in esecuzione mostra chiaramente la dicitura *Host Servizio*.

In `HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\SVCHost` troviamo i *nomi dei gruppi di servizi*. Ad esempio, l'attributo *Camera* $\in$ *FrameServer*, invocabile con `svchost.exe -k groupname`, facendo partire un'istanza associata al gruppo di servizio. Tornando in:

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\FrameServer`

li troviamo, e nella sottocartella `parameters` troviamo i parametri. Un malware può aggiungersi senza troppi problemi a questi gruppi.

Stiamo implementando **segretezzza** e **persistenza.**

### Trojanized System binaries

Sfrutta gli eseguibili di sistema, legato anche alla possibilità di *scalare* i privilegi. Un servizio che parte, lo fa tramite privilegi di *sistema operativo*. Prendiamo l'eseguibile `explorer.exe`, che viene eseguito sempre all'avvio del pc, poichè realizza l'interfaccia dei file visibile quando cerchiamo un file. E' sempre attivo, allora il malware potrebbe modificarlo. Il modo più semplice per farlo è mediante **Detour**, ovvero lasciandogli fare ciò che faceva, ma aggiungendo qualcosa in più.

#### Esempio

```
# FILE ORIGINALE
L0:
    PUSH EBP
    MOV EBP, ESP
L1:
    //fai qualcosa

# FILE DE-TOURIZZATO
L0:
    JMP L2          // lunghezza in byte pari alla vecchia L0
L1:
                    // operazioni legali

//Aggiungiamo L2 che carica una .dll
L2:
    PUSHA           // per evitare di danneggiare il programma originale, salvo tutti i registri sullo stack
    CALL L3         // il codice "detour" non so a priori dove sia rilocato,
                    // alcuni riferimenti "salterebbero"
                    // soluzione: si fa una call poco più avanti al detour che sta girando,
                    // cosi ho un "riferimento"
    NOP             // indirizzo di ritorno della call
L3:
    POP ECX         // non so l'indirizzo assoluto su cui gira, ma il codice si.
                    // metto in ECX indirizzo ritorno NOP, che poteva
                    // essere stato rilocato da windows
    MOV EAX, ECX    // lo metto in EAX
    ADD EAX, 24h    // sommo 24 posizioni (da NOP a .db `malware.dll`,0)
    PUSH EAX        // metto il nome della .dll sullo stack, ora devo caricare LoadLibrary

    ADD ECX, XYZh   // "numero magico", ottengo indirizzo variabile che contiene indirizzo
                    // di caricamento di kernel32.dll; perchè kernel32.dll? perchè contiene LoadLibrary

    MOV EAX, [ECX] // in EAX metto indirizzo di caricamento di Kernel32.dll,
                    // leggendo dalla memoria, non registro. Nota l'informazione di dove parte Kernel32.dll,
```

```
                //  LoadLibrary sarà un certo spiazzamento

ADD EAX, 0ABCh // E' una distanza/altro numero magico, ora so da dove parte LoadLibrary.

CALL EAX       // abbiamo caricato la libreria, senza che essa compaia mai "direttamente".

POPA           // FINE DETOUR: rimetto tutto a posto, ritorno al codice originale come prima

PUSH EBP       // Rifaccio le istruzione saltate nel file ORIGINALE ad etichetta l0:
MOVE EBP, ESP

JMP L1         // salto all'indietro, continua l'esecuzione originale

.db `malware.dll`,0  //stringa a distanza 24h
                     // `malware.dll` sullo stack, e lo carico in LoadLibraryA
```

I programmi saltano ad un certo indirizzo, però Windows può rilocare, e quindi nell'header si mette un riferimento. Il sistema operativo non sa quale punto usa il codice assoluto, bisogna ricavarlo! L'idea è crearsi dei riferimenti (come `call L3` punta poco più avanti). Successivamente, supponiamo di avere, a una certa distanza `24h` dal `NOP` in `L2`, il nome della libreria che vogliamo caricare, ossia `malware.dll`. Cosa mi serve per caricarla? `LoadLibrary`. Come ci arrivo? E' inclusa in `Kernel32.dll`. Devo prendere lei! Questa si troverà ad un certo offset `XYZh`, che salviamo.
Come "conosco" questo offset?
Noi sappiamo la *struttura dell'eseguibile*. Ogni eseguibile ha indirizzo in cui carica le *.dll* che usa, quindi c'è per forza `kernel32.dll`, che non cambia. Possiamo calcolare la differenza, ovvero `XYZh`.

`LoadLibrary` si troverà ad un certo offset `0ABCh` da `Kernel32.dll`. Preso il suo indirizzo di caricamento, eseguiamo il `push` sullo stack. Abbiamo finito il **detour**. La fase finale consiste nel ripulire tutto, ovvero ripristinando le istruzioni *saltate* e ritornando, tramite `jump` all'indietro, al flusso originale.

Ora, facendo ciò, non stiamo **alternado la firma del programma**? Si, e questo ci è di introduzione per altre tecniche. [risata malefica]

### DLL Load-order Hijacking

Persistenza senza modifiche a file o registri del sistema. Prendiamo sempre `explorer.exe`, che usa una certa *.dll*, ma come viene trovata nel sistema? Esistono punti *predeterminati*, ispezionati *in ordine*:

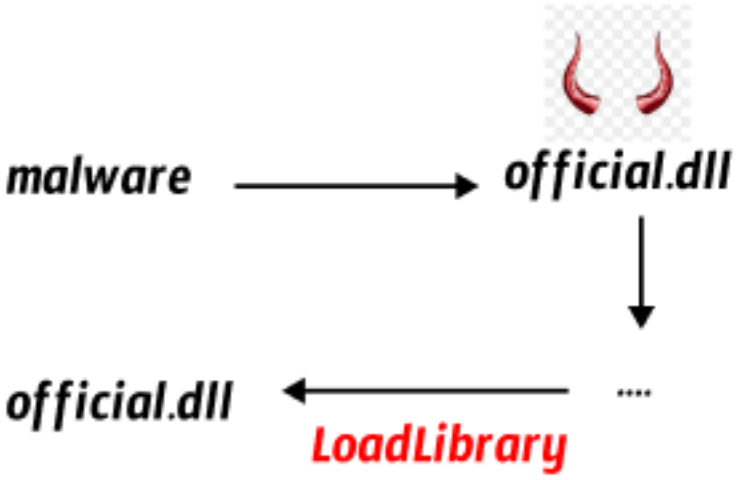
1. Cartella dell'applicazione.
2. Cartella corrente (dove la lancio), $\neq$ cartella dell'applicazione (dove il *.exe* è presente).
3. Invocazione di `GetSystemDirectory()`, che ritorna cartella di sistema.
4. $16-bit$ system directory (preistoria).
5. `GetWindowsDirectory()`
6. `%path%`, variabile di ambiente.

Il *trucco* è chiamare la propria *.dll* malevola **come** la *.dll* ufficiale di sistema, e metterla in qualsiasi punto predeterminato **precedente** a quello della *.dll* ufficiale. *Non viene fatto alcun check sulla signature.* Per *fixare*, i tizi di Microsoft hanno predisposto:

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\SessionManager\KnownDLL` ovvero alcune *.dll* più pericolosette sono **esentate** da questo meccanismo di ricerca. Tra queste c'è *user32.dll*, che dovrebbe essere una di sistema! Basta trovarne una non qui dentro, ma interna ad *explorer.exe* per aggirare il problema. Potrei togliere una *.dll* da questo insieme di chiavi, ma lascerei una traccia. Siamo arrivati in fondo, e non c'è via d'uscita.

Quindi, trovare due *.dll* uguali vuol dire aver piena certezza di avere un malware? **No**, potrebbero farlo anche programmi normalissimi.

Il percorso è del tipo:




Dove sostanzialmente, la prima *official.dll* malevola compie azioni di *detour* e poi continua col suo flusso di esecuzione *legale*.

# 21 dicembre 2023

## Privilegi

È cruciale per il malware ottenere i privilegi per fare il suo lavoro, questo perché il modello di sicurezza di un sistema Windows è basato sul fatto che gli utenti "normali" non possano fare tutto ciò che vogliono nel sistema (in quanto non hanno i permessi per accedere alle risorse del sistema). Ci sono una serie di *account privilegiati* (aventi quindi privilegi superiori ad account normali) che permettono di eseguire operazioni in più rispetto ad un account normale, come l'installazione di applicativi nel sistema.

Si può parlare di livello:

- *utente*
- *amministratore*
- *Nt Authority System*: dà i diritti sulla macchina locale ma anche su tutte le macchine nel dominio in cui è inserita.

L'obiettivo del malware è quello di arrivare al livello più alto possibile, ad esempio per far passare il malware da una macchina ad un'altra.

Tipicamente quando il malware si insidia è perché l'utente ha fatto qualcosa di stupido. Normalmente (o almeno si spera), il malware viene eseguito con privilegi di utenti normale, il suo scopo è quello di realizzare una **escalation di privilegi**.

A questo modello va affiancato un altro.
Windows assegna ad ogni processo in esecuzione **una capacità** (diritto di fare qualcosa) **elementare** (spegnere la macchina ecc...). In **Unix** corrisponde al meccanismo messo in piedi tramite i sistemi che permettono di spacchettare tutte le operazioni che un utente privilegiato può fare in termini di capacità unarie (singole operazioni).

### Descrittori di sicurezza

Per manipolare i *descrittori di sicurezza* si utilizzano opportune API di sistema. Questi vengono descritte come **access token** che descrivono il sistema di sicurezza di un processo o di un thread. Questi possono essere manipolati usando un certo numero di API per modificare il livello di sicurezza, il sistema poi dovrà consentire o meno questa operazione sulla base dei privilegi che l'utente ha associato al processo.

Tutto ciò viene fatto per ottenere privilegi, i quali consentono di eseguire operazioni che, in normali contesti di esecuzione, non dovrebbe essere consentito.

**Come scalare privilegi?**

Se un processo parte, come utente di base, per *aumentare il livello di privilegi* bisogna sfruttare le debolezza del SO (di base non dovrebbe essere possibile). Windows è talmente complesso che si trovano continuamente modi per scalare i privilegi.

È molto difficile cercare di impedire questo tipo di operazioni.

**Dimostrazione**
Prendiamo la nostra macchina virtuale con Windows 10 abbastanza aggiornato e proviamo ad acquisire informazioni sul sistema (esistono vari modi).

Apriamo una shell, come utente normale, e lanciamo un comando presente su tutti i sistemi Windows (ad esempio `systeminfo` che restituisce tutte le informazioni *hardware* e *software*).
Le informazioni ottenute vengono re-indirizzate in un file, che salviamo in locale su Linux (l'attaccante lo ha portato sulla sua macchina). Esistono una serie di strumenti che automaticamente analizzano e cercano le vulnerabilità.

- Il più famoso è `Metasploit` , il quale permette di ricevere l'indirizzo di una macchina Windows e sonda in tutti i possibili modi per entrare nella macchina e scalare i privilegi, tutto ciò in base ad un database di vulnerabilità conosciute. Serve a cercare le vulnerabilità del sistema ma anche per condurre un attacco.
- `wesng` è un dabase fatto bene che elenca le vulnerabilità di Windows. Il file con le informazioni del sistema viene passato in input a questo programma, il quale ci informa sulle vulnerabilità trovate. E' propenso a dare falsi positivi, poichè alcune vulnerabilità potrebbero essere patchate da patch diverse da quelle che si aspetta lui. Con l'opzione `-e` si riescono a ottenere i problemi con exploit noto. Qui ci sono solo le vulnerabilità conosciute, nel dark web si possono trovare vulnerabilità *0 days*, ossia quelle non ancora conosciute. Di vulnerabilità se ne scoprono ogni giorno, sono continue anche le patch di sicurezza. Analizzando il malware si scoprono le vulnerabilità, vedendo in particolare come questo è entrato nel sistema.

## Segretezza

L'obiettivo è infettare una macchina senza che il tecnico o l'analista se ne accorga (in quanto l'user normale è difficile ci faccia caso!)

Ciò è possibile, e la base per nascondere l'attività nel sistema agli analisti è mediante **Rootkit**, ovvero una componente che si installa nel sistema, avente scopo di nascondere l'attività di processi, demoni o traffico di rete, che potrebbero rilevare la presenza di malware. **Il rootkit stesso è un malware, che nasconde un altro malware**. Possiamo catalogare due tipologie:

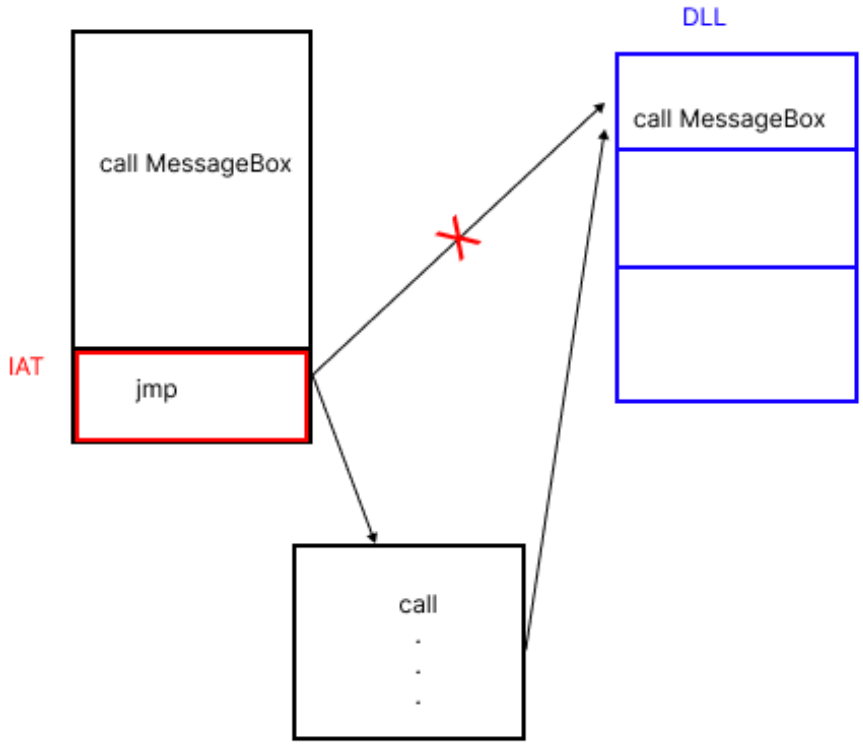
- **user mode**: Lavorano a livello applicativo, senza toccare la parte kernel del SO. E' scopribile tramite applicazioni user mode, usate dall'analista. Facili da implementare, ma facili da scoprire. *Noi osserveremo loro.*
- **kernel mode**: Si installano nel sistema operativo stesso. Gli strumenti user mode sono inefficienti, in quanto il rootkit vi è "superiore". Complessi da implementare e da scovare. Niente analisi black-box, in quanto il tutto è compromesso in partenza. Ne è un esempio l'installazione di un modulo kernel atto a nascondere un processo in esecuzione.

**Rootkit user mode**

Un procedimento è dato da **Import Address Table (IAT) Hooking**. Dato un eseguibile, il quale presenta una `call MessageBox`, questa `MessageBox` sarà dentro una *.dll*, esterna, linkata dinamicamente. Quindi non è dentro l'eseguibile. Nell'eseguibile vi è un'area **IAT**, che tramite `jmp` salta al riferimento presente nella *.dll*, ovviamente quando viene caricata, per sapere dove saltare. Il malware potrebbe pensare ciò: "Un'API pericolosa (come qualcosa che fa vedere i processi attivi) può essere bloccata, evitando il salto alla routine della *.dll*, magari *portando il flusso sul mio codice*, che a sua volta andrà alla routine originale, ma *prima modificherà aspetti del codice per nascondere la sua presenza*."

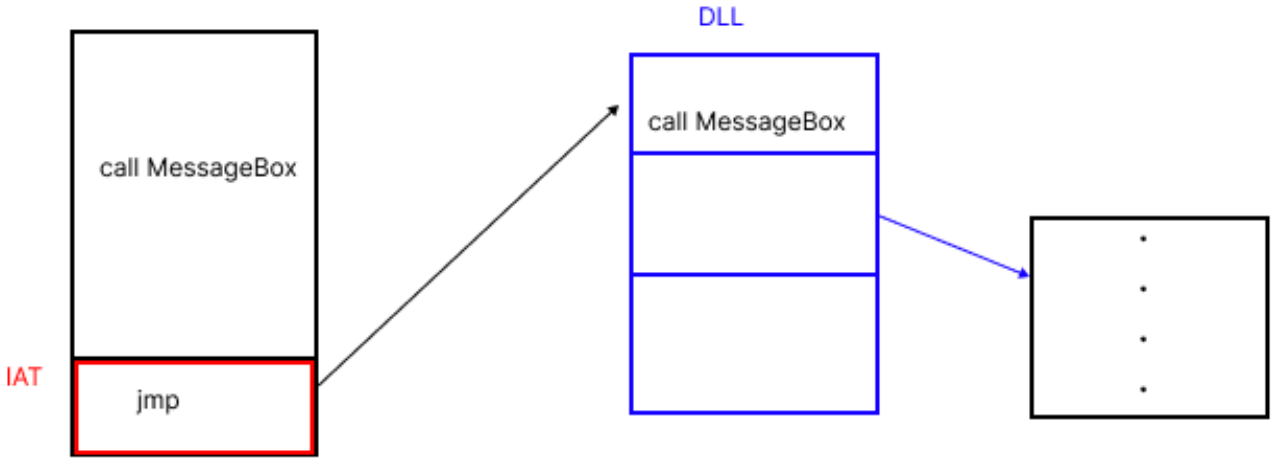
Questo va fatto per ogni processo che potrebbe far riferimento, e quindi è scomoda.

**Può sembrare un *detour*, ma qui la modifica viene fatta nella *IAT*, mentre quello che abbiamo visto lavora nella zona *text*.** E' facilmente individuabile, basta confrontare il tutto con la *firma ufficiale*.




Oggi si usa **Inlive Hooking**:

Il *detour* è presente a livello di *.dll*, modificando proprio la `.dll`. L'idea è visibile nell'immagine:




Magari, prima di ritornare, azzero le informazioni compromettenti. Anche qui si può controllare la firma ufficiale, tuttavia il *detour* può essere svolto sia a livello *fisico* sia in *memoria*. Possiamo lasciare il *file.dll* invariato, e fare il **detour in memoria**. Tutti i modi che permettono di associare al processo in esecuzione la *.dll* può essere soggetto a *detour*.

## Evasione

### DLL injection

Vogliamo inserire codice che *carica* la `.dll` malevola. Ciò include:

1. Prendere il PID del processo vittima (il quale *deve già esistere*, con determinati privilegi di nostro interesse). Ne è un esempio `CreateToolHelp32Snapshot`, insieme a `Process32First` e `Process32Next`, ovvero **iterazioni** per trovare i processi attivi. Concettualmente è come `ps ax` in Unix, da terminale.
2. Ciò ci permette di ottenere un *handle* del processo, con `OpenProcess` (il processo esterno siamo interessati a manipolarlo, ma già esiste).

3. Un processo può allocare memoria ad un altro processo, con `VirtualAllocEx` (tipicamente per scriverci nuovo codice), o scrivere nella memoria di un altro processo con `WriteProcessMemory` (per metterci il codice malevolo), oppure creare un nuovo flusso di esecuzione per quel processo con `CreateRemoteThread` (quindi, ad esempio, **dirottarlo** per fargli chiamare `LoadLibrary` e fargli eseguire la mia *.dll*)

Tutto ciò senza farsi scoprire, perchè opera in memoria. Spesso, la vittima target è `explorer.exe`, in quanto gestisce tante *.dll* e tanti token. L'obiettivo è caricare il codice `.dll`, tramite dirottamento.

### Direct Injection

Troviamo il processo vittima nello stesso modo di prima, quindi prendendo PID, allocando memoria, e scrivendoci. Ora però non carichiamo una *.dll*, bensì installiamo un *detour*. Funziona se il sistema è malconfigurato, come nel caso in cui un processo con privilegio $x$ sia in grado di modificare le pagine di un processo con privilegio $x-1$ (quindi il processo modificato è *più importante* del processo che lo sta modificando). Qui non si caricano *.dll* esterne, bensì il codice viene eseguito nel processo, internamente.

### Process Replacement

Qui la vittima è `svchost.exe`.

L'idea è che sia possibile fare `CreateProcess` per `svchost.exe`, il quale viene considerato privilegiato dal Sistema. Internamente, `svchost.exe` fa dei controlli su cosa può essere fatto o meno. E' però possibile usare il *flag* `CREATE_SUSPENDED`, **creo ma senza lanciarlo** (quindi non salto mai al suo entry point), il processo rimane in attesa.
Chi crea questo processo può manipolarlo, come?

```
CreateProces(..., "svchost.exe",...,CREATE_SUSPENDED);
ZwUnmapViewOfSection(...);
                            // syscall basso livello che opera sul processo creato
                            // butta pagine memoria del processo creato, lo resetto
                            // MA NON RESETTO i suoi privilegi, è quindi un guscio vuoto.
VirtualAllocEx(...);
WriteProcessMemory(...);
                // Abbiamo allocato e ci stiamo scrivendo dentro
SetThreadContext(...);
                // Imposto entry point iniziale di questo nuovo processo
ResumeThread(...);
                // Risblocchiamo/Riavviamo il processo
```

Ricordiamo la differenza tra **syscall**, ovvero l'esecuzione ci permette di passare da *user mode* a *kernel mode*, e l'**API**, la quale permette a due software di comunicare tra loro.
Il passaggio di privilegi, controllato, viene svolto da un *wrapper*. C'è un API che invoca la *system call*. **Le API contengono le *syscall***, tipicamente le API sono di basso livello e user mode, quindi difficilmente in un API troviamo cose come `NTDLL.DLL`. Nella *.dll* appena descritta, troviamo che le API di basso livello iniziano con `Zw...`, quelle di livello poco più alto iniziano con `Nt...`

### Hook Injection

Abbiamo detto che Windows può caricare *.dll* quando viene fatto login, logout, etc... In realtà può anche intercettare, mediante *hook* (deviazioni, rielabola i messaggi), i messaggi di tutti i processi nel sistema. Ciò mi è da chiave per eseguire codice nei processi privilegiati. L'idea è la seguente:

L'utente crea un *evento*, che richiama il *SO*, il quale comporta un *messaggio*, che richiama una *.dll* per la sua gestione, e tale messaggio viene passato al *Thread dell'applicazione*, portando quindi al caricamento della *.dll* Classifichiamo:

- **Local hock**: interni al processo del sistema. Non li vediamo.
- **Remote Hook**, che si suddivide in:
  - **Low-level hook**: *L'applicazione* che installa l'hook deve *rimanere in esecuzione*, se la chiudo, l'hook scompare (scompare il codice per gestirlo). Quindi il codice è nel processo che mette in piedi l'hook. Ci sarà un meccanismo tale per cui il SO, quando deve gestire un messaggio legato ad un hook, risveglierà il processo che ha creato quell'hook.

– **High-level hook**: Usa il meccanismo delle *.dll*;
la singola *.dll* viene caricata dove viene ricevuto il messaggio/evento. *Il malware che lo installa può chiudersi*. E' visibile come un altro modo per iniettare le *.dll*, ma ha anche degli usi specifici, come i **Keylogger**, ovvero si intercettano i tasti premuti sulla tastiera, che alla fine sono eventi. Lo si fa con l'**hook** `WH_KEYBOARD` (con il suffisso `_LL` è di tipo *low level*).
Il malware potrebbe usare: `SetWindowsHookTx(TipoDiHook, funzioneDaEseguire, HandleEventoProcedura, ChiRiceveHook)`.
L'ultimo parametro può essere posto a 0, ovvero consideriamo tutti i thread. La funzione può semplicemente eseguire una routine del tipo: "*Al premersi di un tasto, esegui una funzione che copi il tasto premuto su un file*"

#### APC injection (Asynchronous Procedure Call)

E' possibile eseguire delle *routine* in un altro processo. Cioè chiedo ad un processo: *"quando sei idle, fai questa cosa per me"*. Ovviamente non posso farlo *quando voglio io*, ma solo se vengono chiamate le seguenti API:

- `WaitForSingleObjectEX`, ovvero in attesa di qualcosa (anche un mutex).
- `WaitForMultipleObjectEX`, attendo più oggetti.
- `SleepEx`, dorme.

In queste condizioni, il contesto è già *salvo*, il SO è già pronto a sostituire il processo con un altro. L'idea è che in questi casi, gli facciamo eseguire altri pezzi di codice nel contesto del processo stesso! L'**APC Injection** presenta variante *user mode* e *kernel mode*.

L'API per realizzare ciò, a livello user è : `QueueUserAPC(FunzioneDaEseguire, HandleThread, ArgomentiPassati)`, la quale schedula l'esecuzione di una operazione asincrona, all'interno del processo. Un tipico use case è: `QueueUserAPC(LoadLibrary, hThreadId, "malware.dll")`.

## 9 gennaio 2024

Spesso il codice viene iniettato dentro altri processi. Vedremo come fare ciò. Per ora, ritorniamo su alcuni concetti precedenti.

### Offuscamento delle stringhe

Supponiamo di avere un programma che non riusciamo ad analizzare con Ghidra o Disassembler. In ogni caso, abbiamo il programma, quindi potremmo ad esempio cercare le *stringhe*, che, se non inserite dal programmatore, potrebbero appartenere a qualche libreria usata da lui.

Oggi aumentano i malware scritti con linguaggi interpretati o su macchina virtuale (come Java). Questi però sono più *semplici* da analizzare, in quanto presentano meno tecniche di quelle già analizzati. Dall'altra parte, l'offuscamento delle stringhe risulta più facile.

#### XOR

Una tecnica semplice è l'uso dello **XOR**, byte a byte. Dopo aver scelto una *chiave*, applico l'operazione tra ogni byte e chiave. Lo svantaggio è che le informazioni che vado a cifrare, contengono sequenze di byte facilmente indovinabili, come ad esempio una serie di 0. Se la chiave fosse fatta da un solo byte, e stiamo cifrando un intero file eseguibile, posso dire che il byte che ricorre più frequentemente è la chiave. Oppure, se sto cifrando un *PE*, so che c'è un certo ordine da seguire. Nei malware, si preferisce una variante dello **XOR**.

```
if (c[i] != 0)   //evitiamo le occorrenze di 0
    {c[i] XOR k[i]}
else { c[i] }
```

L'ambiguità è che se `c[i] == k[i]`, ottengo 0. Allora la versione corretta sarà:

```
if ( c[i] != 0 && c[i] != k[i] )
    { c[i] XOR k[i] }
else { c[i] }
```

Questa è la variante più *diffusa*. Se conosco però il testo in chiaro (o qualcuno dei byte in chiaro), lo XOR espone una parte della chiave, o più.

Se so che qualcosa inizia con un certo byte (come il testo in chiaro), per ogni possibile byte creo una tabella con varie $k$, e vedo quando c'è una corrispondenza. Posso applicare tecniche diverse dallo XOR, come shift o rotazioni, purchè reversibili. Esistono programmi che cercano stirnghe di offuscamento negli eseguibili.

#### Base 64

Supponiamo di avere sequenza di byte, avente dimensione *8 bit*. Questi 8 bit danno vita, per ogni byte, a 256 valori. Tuttavia, non tutti sono rappresentabili mediante ASCII. Quindi bisogna convertire queste sequenze di informazioni in un formato composto da lettere stampabili. Con *base_64*, dividiamo in gruppi da 6 (partendo da *sinistra*). Ogni gruppo è 1 di 64 possibili valori ($2^6$), abbiamo 64 caratteri stampabili. Possiamo stampare:

- Lettere da [A... Z], 26 lettere.
- Lettere da [a... z], 26 lettere.
- Numeri da [0... 9], 10 lettere.
- Caratteri "+" e "/".

Se partiamo da sinistra, alla fine, verso destra, potremmo avere qualche problema. Potremmo aggiungere 1 o 2 caratteri di *padding*, che corrisponde al simbolo "=", e serve per arrivare ad una lunghezza finale di bit che sia multiplo di 8. Possiamo vederlo con `echo -n "[testo]" |base64`.

Una prima azione per *ostacolare* l'analista, offuscando le stringhe `BASE_64`, è omettere i caratteri del padding, o cambiare l'ordine della stringa. Possiamo anche cambiare il set di caratteri.

```
Rappresentazione ASCII:                 A           B           C
                                        |           |           |
Rappresentazione binaria:            01000001    01000010    01000011
                                       /    \      /    \      /    \
Suddivisione in gruppi da 6 bit:    010000  01  0100  0010  01  000011
                                    |----|  \------/  \------/  |----|
I 4 valori dedotti:                 010000   010100    001001   000011
                                      \/       \/        \/       \/
Il valore decimale:                   16       20         9        3
                                      |        |         |        |
Il valore codificato:                 Q        U         J        D
```

#### Funzioni crittografiche

Ci nascondiamo mediante crittografia, introdotta nel programma. Ciò ci permette anche di modificarlo (magari cambiando delle costanti), per evitare che funzioni come AES vengano risolte facilmente. Tuttavia, se l'algoritmo è crittograficamente semplice, mi espongo al rischio che sia possibile rompere il cifrario e recuperare i dati. E' anche possibile cercare le API correlate alla crittografia, spesso identificate con *crypt*, *cp*, *cert*. A lezione si è visto un programma chiamato ***PED***, che presenta un plugin *crypto-analyzer*.

### Offuscamento del codice

#### Tool per l'entropia

L'entropia riconosce parte di codice compressa, ma anche criptata, perchè in entrambi i casi ciò che viene prodotto è altamente casuale, altrimenti si potrebbe attaccare.

#### Debugger

Posso, ricostruendo il flusso dei dati, vedere cosa fanno le funzioni. Oppure *detouring*, modificando l'eseguibile in modo che faccia ciò che vogliamo noi.

#### Shellcode

Deve essere inserito in un altro processo in esecuzione, il quale deve essere funzionante.

Prendiamo un processo generico, a cui vogliamo inserire un'altra parte di codice, tale che essa venga eseguita in un certo punto.

Come scrivo quello codice, cioè questo *shellcode*? Quando lo analizzo, cosa devo aspettarmi?

Lo shellcode **non può** fare affidamento sull'essere caricato in un certo indirizzo di memoria, in quanto *non può avere alcun supporto da parte del sistema operativo*. Gli indirizzi che troviamo in programma/codice macchina sono:

- `call fn`, chiamata a funzione.
- `jmp label/indirizzo`, jump ad etichetta o indirizzo.
- `mov eax, [edx]`, metto in eax l'indirizzo di *memoria* contenuto in altro registro. Oppure `mov eax,var`, dove copio il valore.

Classifichiamo inoltre:

- Indirizzo **assoluto**: a 4 byte, detto virtuale o lineare e rappresenta l'indirizzo di *dove voglio saltare*, o dove voglio invocare. Se cambio la posizione di una chiamata, dovrò cambiare anche l'indirizzo assoluto.
- Indirizzo **relativo**, è un *offset* rispetto all'*instruction pointer EIP*. Non è detto che cambiare un indirizzo comporti ricalcolare anche l'indirizzo relativo. (Ad esempio, se faccio uno shift di tutto il codice, l'indirizzo relativo non viene alterato.) Questo tipo di codice è anche detto *codice indipendente dalla posizione* **PIC**.

Per codice che punta ai dati, si usano solo **indirizzi assoluti** (come nell'esempio `mov eax, var`). Infatti non posso dire *"dammi indirizzo cella a x posizioni rispetto ad EIP"*.

Con `mov eax, [edx]`, visto che non specifichiamo nessun indirizzo, siamo in un contesto **PIC**.

Quando usiamo `call MessageBoxA`, esso è un indirizzo assoluto. Saltiamo a *wrapper* che ci fa saltare a una `user32.dll` contente ciò che ci serve. I programmi di Windows sono *rilocabili*, e sfruttano tabelle di rilocazione.

In Unix/Linux, si usa codice PIC sopratutto per le librerie dinamiche, perchè il problema è dovuto al fatto che non so quante e quali ne userà l'eseguibile.

In Windows, le *.dll* hanno indirizzo predefinito (ciascuna ha un proprio indirizzo di base) e devono essere rilocate se situate in un indirizzo differente, ma ciò è costoso; se è evitabile è meglio. I **programmi caricano le *.dll* sempre agli stessi indirizzi per arginare ciò. Solo se due *.dll* hanno stesso indirizzo vengono rilocate dal S.O., altrimenti no.**

Supponiamo di avere *messageBoxA* in *user32.dll*. Abbiamo centinaia di processi nel sistema, non è che ogni sistema si ricarica una nuova istanza della stessa *.dll*, ma usano le stesse pagine fisiche. La stessa pagina fisica viene mappata su spazio di indirizzi virtuali dei vari processi, tramite *Page Table*. Processi diversi *potrebbero* prendere queste pagine fisiche e caricarle ad indirizzi virtuali o lineari differenti, arrivando però alle stesse pagine fisiche. Il sistema operativo, per efficienza, usa pagine fisiche e stessi indirizzi lineari, *in modo da non rilocare* e non sprecare memoria fisica. Questo perchè altrimenti dovrei rilocare il codice. Ecco perchè Windows carica la *.dll* di sistema in un posto predefinito, perchè non si basa su codice *rilocabile*, mentre Linux è PIC, e la pagina fisica o codice non deve mai essere toccato.

Tornando allo **shell code**, visto che non so dove verrà caricato, deve essere per forza **PIC**. Se deve usare una API, può assumere che essa sia ad un indirizzo prestabilito.

Attenzione: questo indirizzo particolare dipende dalla versione del S.O. (magari in Windows 10, `MessageBoxA` viene caricata ad un indirizzo diverso rispetto Windows 8), lo stesso se si aggiorna la versione di `user32.dll`.

Ricordiamo l'esistenza di `LoadLibrary` e `getProcAddress`, che anche se ci permettono di avere indirizzi di API esterne, esse stesse sono API, e quindi presentano lo stesso ragionamento appena esposto.

## Come scrivere shellcode

L'assemblatore più comune è **nasm**, con disassemblatore lineare **mdisasm**, reimplementazione di *masm* (di Microsoft).

Apriamo `vi es1.asm`.

Potrebbe non funzionare su altre macchine.

```
BITS 32
//voglio codice machine independent, devo capire dove mi trovo
    sub esp, 20h  //riservo 32 byte sullo stack, mi serve spaizo
    xor edx, edx  // lo azzero, edx è 0, metto gli 0 sullo stack
```

```
    call l0       // chiamo l0 (mette su stask indirizzo ritorno, ovvero l1)
l1:               // è offset rispetto EPI corrente, non indirizzo assoluto
    db `Hello`,0  //stringa da stampare
l0:
    pop edi
    push edx         //parametri richiesti da MessageBox
    push edi         //push della stringa, titolo finestra
    push edi         //msg da stampare
    push edx
    mov eax, 753d1230h; MessageBox  //invoco messageBox, a destra metto indirizzo preciso
    call eax
    mov eax, 754f9650h; Exit Process   // indirizzo API, in kernel32.dll
    call eax                           //programma termina
```

- `nasm -b32 es1.asm` lo assembla, e produce `es1.obj`
- `ndisasm -b32 es1.obj` ritorna i byte corrispondenti al testo. Le stringhe le vede come istruzioni. Notiamo il codice operativo `E80D`, è un riferimento ad indirizzo relativo.

## 11 gennaio 2024

Proviamo ad aprire uno shell-code con Ghidra, ovvero `lez_27.bin`. Ghidra non sa cosa farci, **perchè non è un file eseguibile**, **non ha una testata**, dobbiamo dirgli noi cosa è. Il formato è *raw binary*, l'architettura/linguaggio non è nota a priori, se siamo sicuri sia un *intel x64 compilato con gcc* allora specifichiamo questo. Poi iniziamo l'analisi, veloce perchè Ghidra non capisce nulla, in quanto la sua analisi parte dall'entry Point, espressa proprio nella testata dell'eseguibile che qui non c'è! Ci mettiamo ad offset 0 e facciamo *disassemble*. Ciò che troviamo è qualche operazione con lo stack, e un jump a `01c`, in cui è presente l'istruzione `FLDZ`. Questa viene eseguita perchè una delle prime operazioni svolte è quella di capire la collocazione in memoria, ovvero ricostruisce l'indirizzo a cui è caricato. Ciò che fa è `push + 0.0 onto the FPU register stack` , ovvero fa una push sullo stack dei registri floating point.

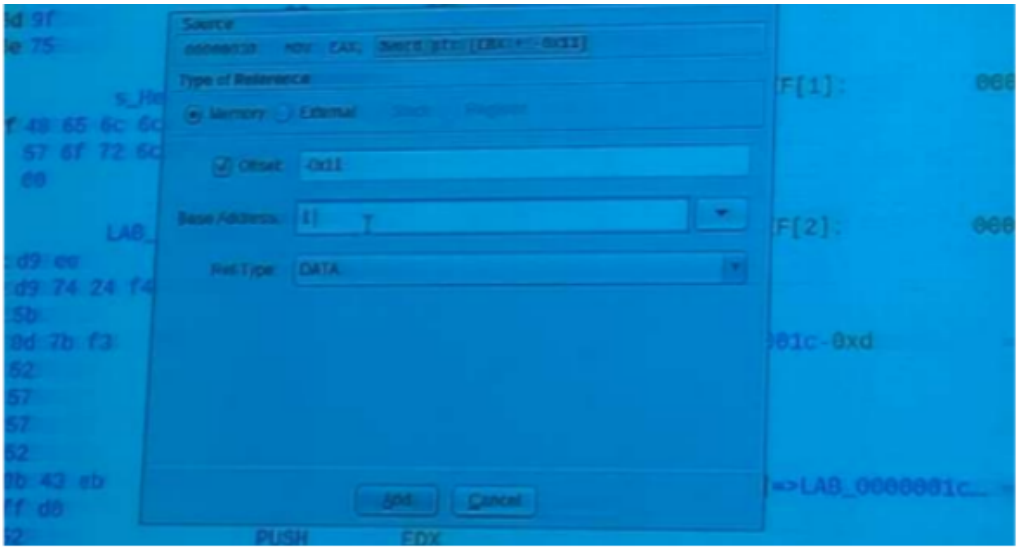
Ancora non ci è chiaro il quadro delle operazioni, allora vediamo l'istruzione dopo, `FNSTENV`, che presenta anche una variante `FSTENV`, sono abbastanza simili, entrambe salvano il *contesto di esecuzione del coprocessore* sullo stack user mode. Normalmente, non viene salvato il coprocessore matematico, perchè se ci fosse un'interruzione il coprocessore non verrebbe modificato, e quindi non c'è bisogno di salvarlo. Se passo ad un altro processo, che non usa il coprocessore matematico, non ho bisogno di salvarlo, lo lascio li. Salvare il contesto del coprocessore è costoso. **Lo salvo se passo tra due processi che lo usano!**

Questa operazione ci interessa perchè tra le istruzioni si ha l'indirizzo dell'ultima operazione floating point usata dall'FPU, ovvero `FPUInstructionPointer`, ad offset 12 dalla fine della struttura `DEST`, e quindi a 12 dall'inizio dello stack, perchè lo stack al contrario.

Torniamo al codice:

```
FNSTENV[ESP + -0xc]  // ovvero sottraggo proprio $12$.
LEA EDI[EBC + 0x1c]  // possiamo aiutare Ghidra specificandogli
                     // l'offset rispetto l'inizio di `LAB_0000001c`
```

Come "aiuto" Ghidra?, tasto destro sul `codice -> references -> add references`, per far apparire la seguente finestra, e mettere come `base address` proprio `1c`



Per l'analisi col debugger, bisogna creare uno *shellcode_launcher*, che inietta lo shell-code da qualche parte. Il libro ne fornisce uno.

Apriamo *shell di comandi*, e compiliamo `shellcode_launcher.exe -i lez-27-01.bin`. Il tutto crasha, può essere un problema di Windows Defender oppure un qualche errore da meglio definire.

Il debugger deve aprire lo *shellcode_launcher*, il quale carica poi quel binario.

Specifichiamo `command file` mettendo `lez-27-01.bin`, ovvero il passaggio di parametri visti come prima. OllyDBG ci dice che la memoria ad indirizzo $x$ non è più leggibile. Vorrei fare debug del launcher, e fermarmi esattamente quando sto passando il *.bin* Chi ha creato il launcher (non il debugger!) ha pensato a questo, e mediante `-bp`, è possibile generare un breakpoint proprio prima di questa azione. Andiamo nelle opzioni di debugging, eccezioni, ed essere sicuri che `INT3 breaks` sia deselezionata, perchè questa eccezione *viene generata dal programma*. Adesso il debug si ferma proprio prima di questa azione, e con `F7` possiamo analizzare cosa fa rompere il tutto. L'errore è a `001E012D`, associata al registro EAX (che mostra un valore in rosso), ovvero la `MessageBox` sta a `77381290`, e non dove abbiamo detto staticamente noi. Perchè fa questo? Perchè è stata modificata la `.dll` con un aggiornamento, oppure per qualche caso particolare è stato rilocata.

Perchè siamo stati sicuri che fosse la MessageBox il problema? Perchè nell'analisi con Ghidra avevamo capito che sarebbe apparsa una finestra *Hello World*.

Possiamo sostituire **EAX** = `753d1230` e ci mettiamo `77381290`, ed adesso funziona tutto.

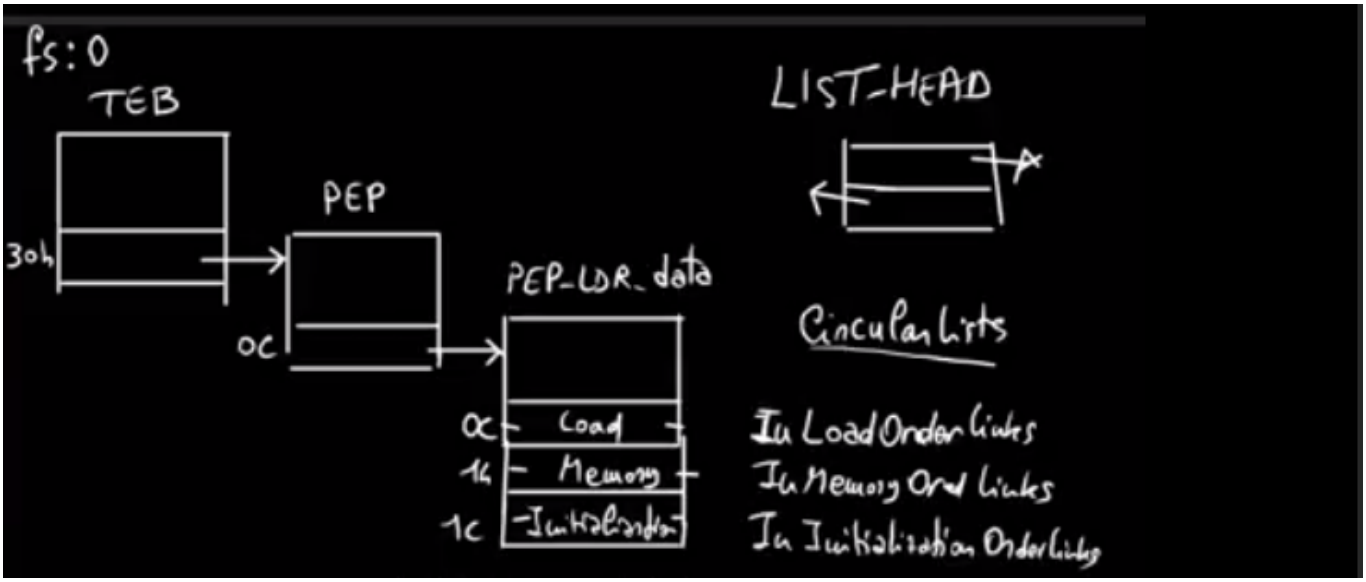
Viene mostrata anche un'altra versione di shell code, robuta anche con gli indirizzi che cambiano, infatti funziona già da subito perchè più moderna.

## Problema dell'indirizzo a priori

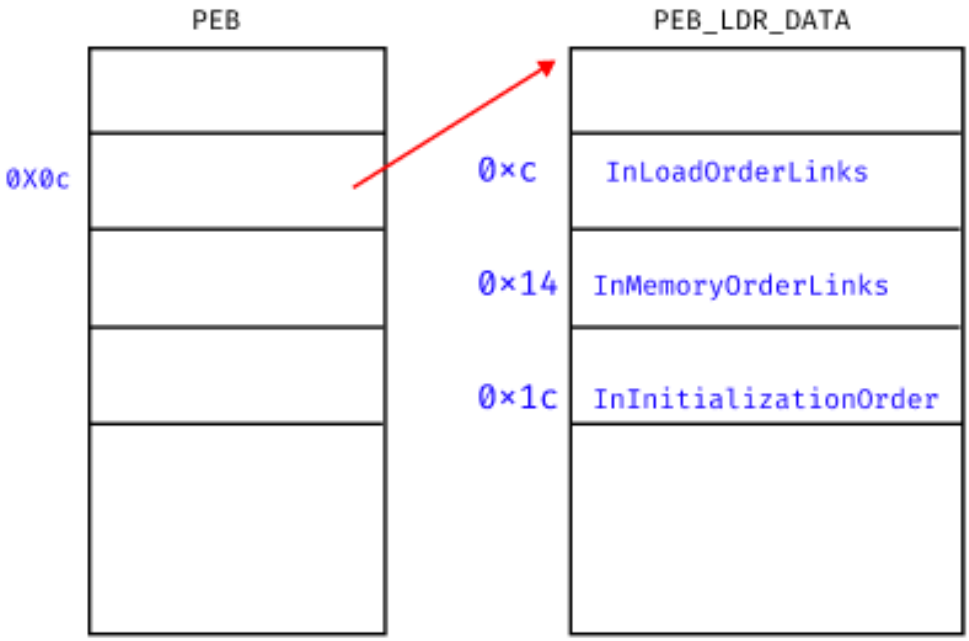
Ritorniamo al discorso API: Due molto usate sono `LoadLibrary` e `GetProcAddress`, situate in `Kernel32.dll` e `KernelBase.dll`, il problema è sempre quello visto: dobbiamo sapere dove si trovano etc...

Tutte queste operazioni non sono ben documentate, però c'è un sito che spiega le strutture interne del sistema, non contenute dalla documentazione ufficlale. Il sito è: http://undocumented.ntinternals.net, ben noto a chi scrive *malware*.

Facciamo un passo indietro: Ogni thread ha descrittore `TEB`, il cui indirizzo è ottenibile mediante registro `FS:0`. Se vediamo la struttura `TEB`, vediamo molti byte reserved (aka: *non te lo dicoh!*). all'offset `0x30` c'è un puntatore alla `PEB`. Nella `PEB` troviamo, ad offset 12 (`0x0c`), `PEB_LDR_DATA`, un puntatore.




Vediamo questa `PEB_LDR_DATA`, con molti campi reserved, e troviamo List Entry`inMemoryOrderModuleLinks` (l'unico documentato). `PEB_LDR_DATA` contiene una serie di campi che servono a descrivere la lista delle DLL caricate.




Ad offset `0xc` c'è `InLoadOrderLinks`.

Ad offset 20, ovvero `0x14` c'è `InOrderModuleList`.

Ad offset `0x1c` c'è `inInitalizationOrderModuleLinks.`

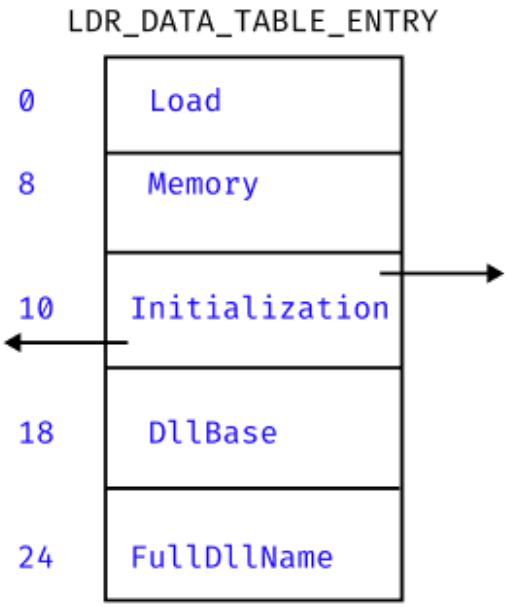
Sono tre **liste doppie circolari ordinate in maniera differente**, contengono le `.dll` caricate nel processo.

- La prima ordina in base a *quando le ho caricate.*
- La seconda ordina *in base alla memoria.*
- La terza in base all'inizializzazione. Le `.dll` possono essere ricorsive (una chiama l'altra), e quindi in qualche caso l'ordine di caricamento è diverso dall'ordine di inizializzazione.

Lo *shell-code* preferisce quella in Order (la terza), il quanto l'ordine è del tipo:

1. `NTDLL.dll`, interfaccia syscall.
2. `Kernelbase.dll`, per LoadLibrary.

"Navigando su `0x1c`", vediamo la struct di `_LDR_DATA_TABLE_ENTRY`, che contiene l'indirizzo base a cui viene caricata la *.dll*. Ognuna descrive una `.dll`, e ogni campo *avanti* e *indietro* punta a strutture di questo tipo.




Tutto questo per ricavare indirizzo di base di `KernelBase`.

Nel launcher c'è anche una funzione che fa l'hash, `HashString` serve per capire quale libreria usare, senza mostare la stringa che serve.

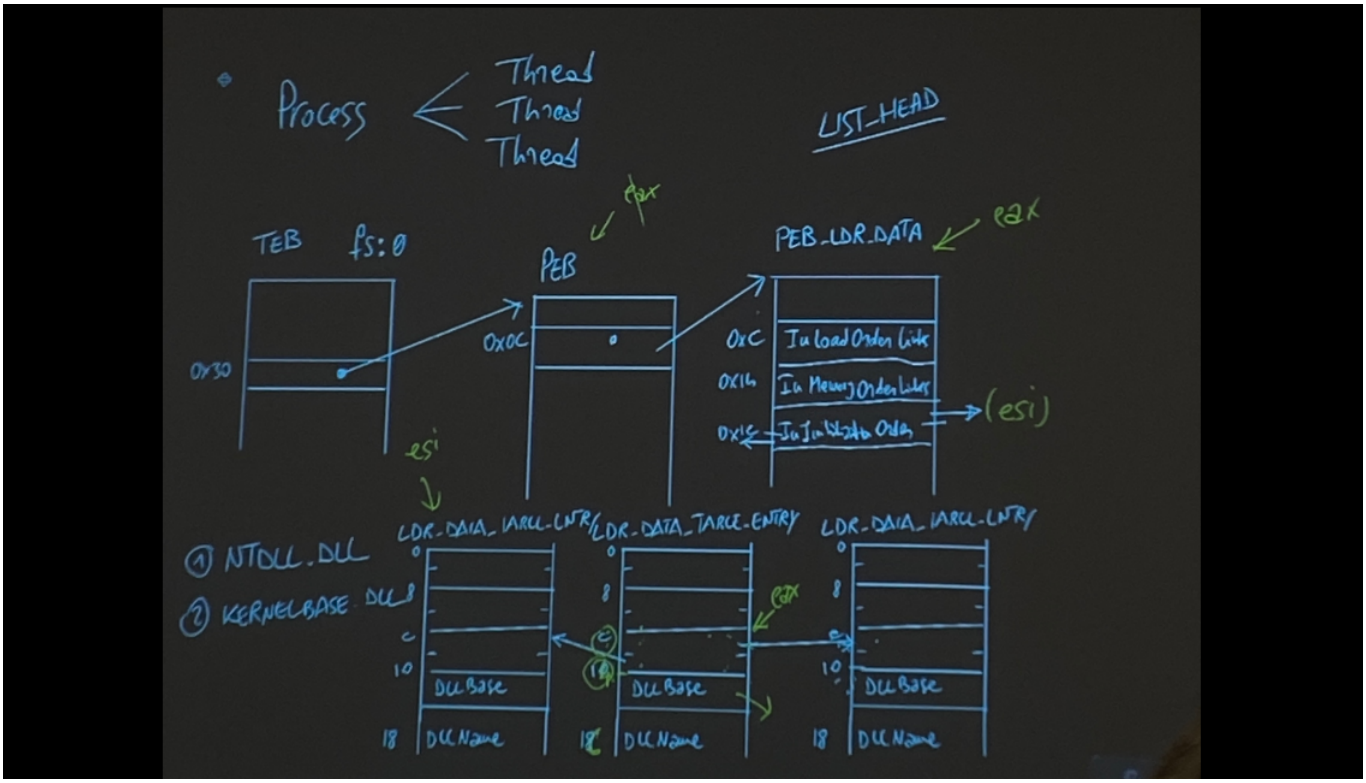
# 16 gennaio 2024

## Analisi dello shell-code

Lo shell-code punta ad arrivare a `.dll` come `LoadLibrary` (per caricare la libreria) e `getProcAddr` (per trovare l'indirizzo dell'API di nostro interesse). Ogni thread, tramite suo registro segmento, può accedere ad area **TEB (Thread Execution Block)**, da cui ricaviamo pointer a **PEB (Process Execution Block)** (che descrive il processo), da cui ricaviamo **PEB_LDR_DATA**, per trovare le `.dll` usate nel processo. Ci sono sicuramente due `.dll`:

- `NTDLL.DLL` (libreria dinamica per avere interfacce per le syscall del sistema operativo)
- e successivamente `KERNELBASE.DLL` (funzioni base offerte dal kernel).

Lo shell code usa, tra le tre liste in foto, la terza lista (non documentata, ma *ordina in base all'inizializzazione*). Ogni lista ha due puntatori (avanti e indietro).




Quando SO carica la `.dll` in memoria, lo fa con memory mapping. Quando troviamo indirizzo base della `.dll`, dobbiamo procedere con la ricerca dell'indirizzo dell'API. Normalmente si fa con `getProcAddr`, ma essa stessa è un'API. Nella testata della `.dll`, cerchiamo di ricostruire ciò che ci interessa. C'è differenza tra eseguire shell code in un ambiente *x32* o *x64*, perchè oggi nel caso *x64* possiamo avere più formati nella testata PE. Usiamo `PE bear`, l'obiettivo è **trovare l'indirizzo dell'API** `LoadLibrary`.

**Come trovo indirizzo nel file .dll?**

- Si parte da testata **DOS HEADER** ad offset 0. Ci interessa l'ultimo campo, ad offset fisso `3c` troviamo `New Exe header = 100`, dove `100h` è un **Relative Virtual Address**, ovvero il *prossimo* header si trova ad un *indirizzo base* (`DllBase`) dell'attuale `.dll`, spiazzato di `100h` (ovvero 256), cioè il successivo PE. Vi troveremo delle *signature* per farci capire che si tratta di una testata.
- Vedendo la *sezione* `File Hdr` (facente parte sempre della testata), troviamo che il primo offset è 104, mentre a 118 inizia `Optional Hdr`. A tale offset, troviamo "magic", ovvero "*magic number*". In questa sezione ci interessa **ExportDirectory**, che sta a spiazzamento `70h`, ovvero 112 decimale. (Trattasi di un altro **RVA**, in una vecchia versione l'offset era 96 decimale.). **Allo shell code questo interessa!**

*Quindi, preso questo RVA e sommato all'indirizzo di base in cui è caricata la DLL si ottiene la tabella degli exports della DLL.*
Lo shellcode deve adattarsi all'ambiente di esecuzione, e deve complicarsi di pari passo al sistema operativo. Uno shell code può, prima di cercare la cartella `export`, vedere il numero `magic` per discriminare quale dei due offset appena visti dovrà usare.

**Trovare la ExportTable**

Carichiamo l'indirizzo di una `.dll` generica (non è detto sia `KernelBase`), sommiamo `3c` (leggiamo ultimo campo testata, ci dice dove sono le altre), prendiamo il primo byte della nuova testata, e leggiamo il magic number, ci chiediamo se sia `NT64` o meno. Se lo è (*magic number* = `20b`), sommiamo 112 a tutto il resto , altrimenti (*magic number* = `10b`) sommiamo 96 byte. Ciò che ottiamo è un **RVA, è offset, non indirizzo**. Per ottenere l'indirizzo sommo dove è caricato. Il formato di `export table` ce lo mostra PE bear, contiene due campi:

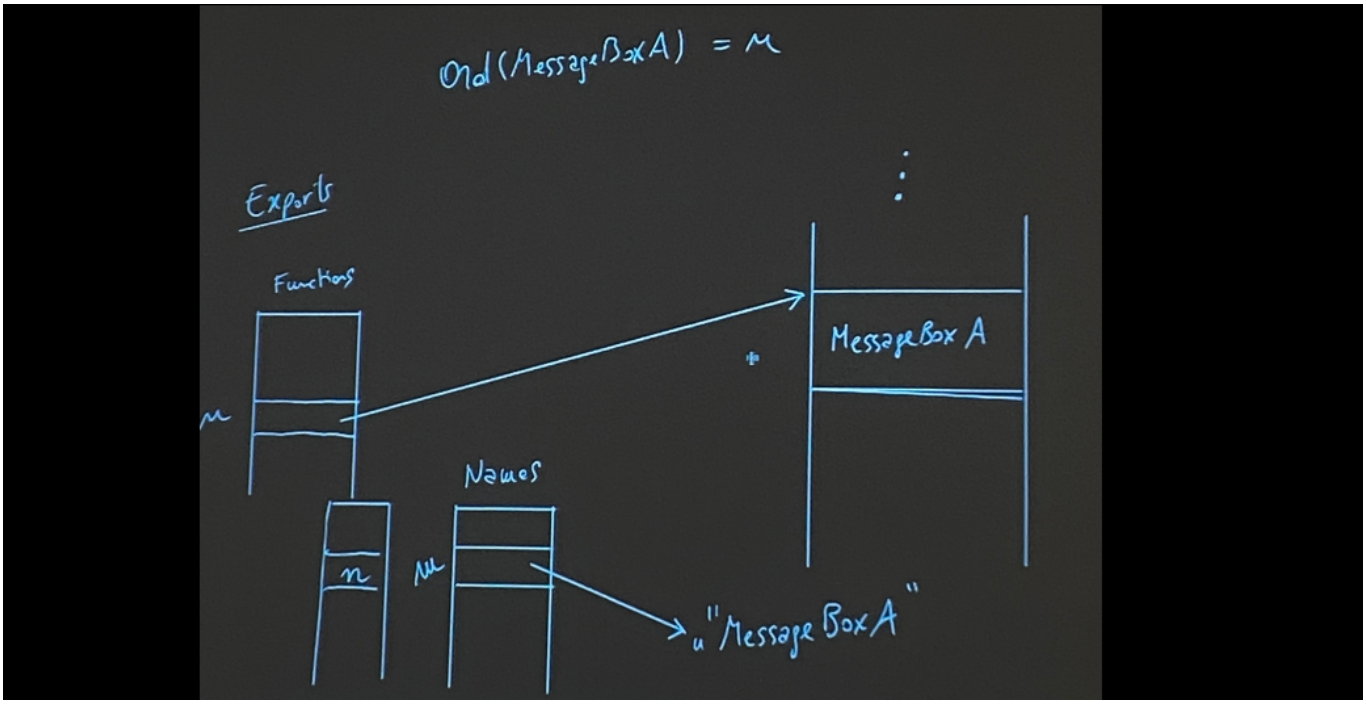
- numero delle funzioni API della dll
- numero di nomi della dll

### Trovare indirizzo di un'API

Nella dll in esame, ogni API ha un nome, quindi i due campi hanno stesso numero. Non è detto sia sempre cosi, se non hanno nome si usa il numero d'ordine (detto *ordinale*). Abbiamo altri tre campi address, che sono tutti RVA:

- `AddressOfFunctions`: vettore che elenca RVA iniziali di ogni API
- `AddressOfNames`: RVA iniziali delle stringhe che codificano i nomi delle API.
- `AddressOfNameOrdinal`: fornisce collegamento indiceNome - indiceOrdinale.

Dalla tabella Export possiamo arrivare ad una tabella di funzioni, cioè una sequenza di RVA, che codificano gli indirizzi iniziali delle nostre API. L'indice di MessageBox in questa tabella è l'*ordinale* di MessageBox, un numeretto appartenente a `AddressOfFcuntions`. All'indice $n$ troveremo MessageBox. Il problema è che non conosciamo tale numero, allora entra in gioco il secondo vettore, `AddressOfNames`. Qui avremo un riferimento a `MessageBoxA`,*Names* è ordinato secondo un ordine alfabetico, per velocizzare la ricerca. Ciò che otteniamo non è l'ordinale $n$, bensì l'indice $m$ nel *vettore dei nomi* di `AddressOfNames`. Ci viene in aiuto il terzo vettore `AddressOFNameOrdinal`, dove in corrispondenza dell'indice $m$ troviamo proprio il numeretto $n$.




Il codice dello shell-code ritorna errore se non trova l'API. Nel ciclo, sommiamo sempre la base, per trovare l'indirizzo stringa, poi calcoliamo `hastString`, mettiamo il risultato in `eax` e confrontiamo con `esp + 0x28`, ovvero quello che stiamo cercando. Se non l'abbiamo trovata, torniamo all'inizio di `search_loop`, decrementiamo `ecx` (scansione dal basso) finchè non lo trova. Quando lo troviamo, `ecx` ci dirà posizione nel vettore di stringhe, prendiamo RVA dei NameOrdinals, e la base, per trovare l'ordinale della funzione (che va di 2 byte alla volta, prima erano 4). Lo stesso ragionamento viene fatto anche per RVA delle functions. Quando ho completato, ritorno tutto su `eax`. Nel caso di `ExitProcess`, su PEbear c'è campo `Forwarder` che ci reindirizza in `NTDLL.RtlExitUserProcess`, cosa che lo shell-code non osserva nelle vecchie versioni. Altra soluzione è usare `exit`, che non ha questo forwarding.

Gli shellcode sono tipicamente codificati, ciò viene fatto perchè lo shell-code viene contenuto all'interno di programmi che devono poterlo estrarre ed iniettarlo. Tipicamente questi programmi sono in un linguaggio interpretato.

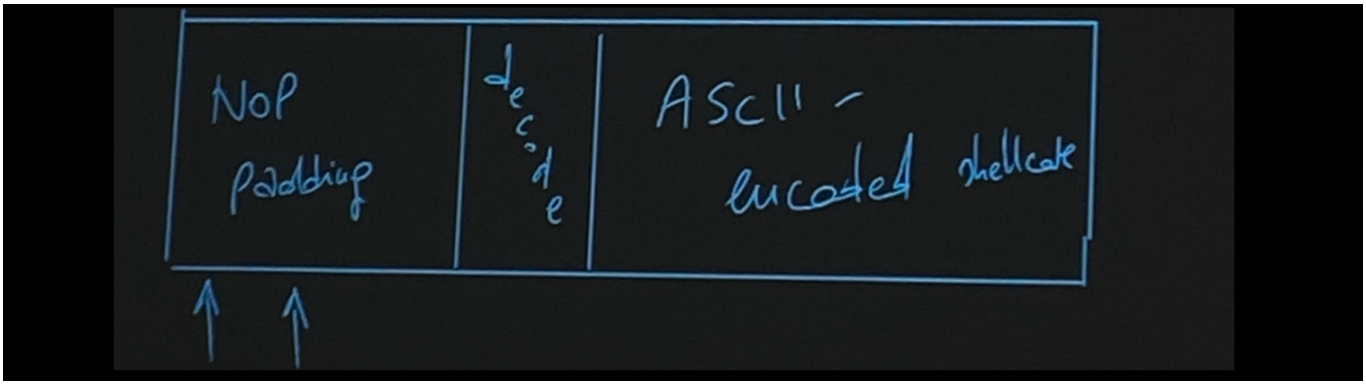
### Uso dei buffer overflow

Gli shell-code possono anche forzare dei **buffer-overflow**, visti come input dell'utente. Vorremmo basarci su `strcat`, `strcmp`, etc... così da sfruttare vulnerabilità legate al buffer overflow. Se buffer allocato sullo stack, posso riscrivere zone in indirizzi di ritorno. L'input è di tipo *testo*, perchè ci si aspetta che l'utente metta un testo. Chi è che decodifica il testo? Nel programma non c'è nulla, allora è lo shell-code che si auto-decodifica. Lo shell-code deve essere anche eseguibile. I problemi sono:

- in codice binario è facile avere degli 0, che però sono assimilabili alla fine di una stringa.

- Le sequenze di byte dovrebbero essere tradotte in codifica Ascii.
- Lo shell-code assume l'inizio al byte 0, ma se forziamo il buffer-overflow, non sappiamo dire il punto esatto, nè la situazione dello stack nè altro.

Conviene mettere una serie di NOP (non `90h` perchè maggiore di $128_{10}$) all'inizio, per realizzare un padding. Non avendo il controllo esatto di dove inizierà l'esecuzione, mettiamo queste NOP in modo da poter andare sempre all'inizio vero e proprio dello shell-code. Poi abbiamo la parte di *decodifica*, in quanto lo shell-code non è in binario *pulito* (contenente 0 e caratteri non stampabili), infine c'è *ASCII-encoded shellcode*, che verrà decodificato per poi saltarci dentro.




Nel decode, dobbiamo produrre solo caratteri stampabili. Per il padding non possiamo usare `90`, poichè non stampabile. Ad esempio, `inc reg` è istruzione macchina che è anche stampabile, ed è un'istruzione di un byte. Tutto deve essere stampabile perchè devo passare l'intera stringa, e quindi deve essere tutto stampabile.

# 18 gennaio 2024

## Vulnerabilità in rete

Il malware moderno è complesso, gli esempi visti sono casi *base*. Il malware fornito per il progetto non è estremamente complesso, ma presente molte righe di codice, suggerendo un approccio ordinato. Il malware cambia, ma le tecniche non cambiano di molto, anche se bisogna rimanere *aggiornati*. Il malware si appoggia su protocolli semplici e diffusi, come `HTTP` (e la variante `HTTPS`) o `DNS`, i quali generano enormi volumi di traffico. Se, ad esempio, una pagina come Bing venisse compromessa, e il server dovesse rispondere a tutto, potremmo iniettare, come commento in `HTML`, una codifica in *base64* di un malware. Questo meccanismo è difficile da tracciare, un firewall non controlla questi aspetti (richiederebbe molto tempo). Si può, invece, intercettare l'attività locale del malware. In merito al `DNS`, noi diamo il nome del dominio, e lui ritorna l'indirizzo IP. Compromettere un server `DNS` ci permetterebbe di ottenere informazioni normalmente non ottenibili. Anche qui, il traffico è elevato, e non può essere controllato. Questo ci fa capire che un *controllo globale sulla rete non è fattibile*.

Supponiamo di avere un server, che compie operazioni legate al NAT, e fa da tramite tra l'esterno e una rete locale. Qui possiamo mettere un controllo. Solo le connessioni uscenti possono ricevere una risposta, entrante nella rete. Se il malware fosse già dentro? Il meccanismo diventerebbe invalido. Ciò capita molto spesso, introducendo pennette usb, o scaricando file nei singoli pc.

Altri meccanismi di protezione, oltre al Firewall appena citato, sono *Antivirus* e **Intrusion Detection System IDS**. L'IDS osserva traffico su rete locale, per riconoscere se le attività sono legittime. Una regola comune è:

- Viene aperta connessione a sito web, che però non è mai stato interrogato da un DNS mediante query. Ciò vuol dire che tale indirizzo era inglobato da qualche parte.

Queste attività, per essere riconosciute, devo però compiersi, quindi l'IDS non *previene* queste azioni. Ciò ci permette di introdurre gli **Intrusion Prevention System IPS**. Ne è un esempio classico quando, scaricando un file di Internet, ci dice che tale eseguibile non è sicuro. Tuttavia, questo alert, nel caso dei browser, viene mostrato per qualunque file scaricato, quindi perde un po' di senso. Esistono meccanismi più rigidi.

## Meccanismo dei packer

Abbiamo visto come, prendendo il malware, lo si possa *codificare* tramite packer, nascondendo il suo contenuto. Ovviamente, deve esserci anche l'operazione opposta, per ricostruire il file originale. Oggi il malware usa crittografia

forte, con algoritmi consolidati come *AES*. Qui la difficoltà potrebbe essere rintracciare la chiave, ma spesso neanche il packer stesso include la chiave di decofica. Il packer potrebbe, ad esempio, cercarla tramite forza bruta (e quindi la chiave non può essere troppo lunga). Potrebbe farlo anche l'analista? Certo, ma lui non sa se la chiave sia già presente o meno, deve quindi cercare prima nel packer e poi applicare la forza bruta.

#### Packer con chiave remota

Altra variante potrebbe vedere il packer che, a runtime, interroga un server per farsi dare la chiave. In questo caso, noi non otteniamo tale informazione finchè non eseguiamo il malware.

#### Malware suddiviso in blocchi

Packer lavora a fianco del malware, il quale è diviso in blocchi, e di cui il packer decofica di blocco in blocco.

#### Protezione del packer

L'obiettivo, spesso, è proteggere una macchina virtuale, la quale emula architettura o processore non standard e non canonici. Magari, questa VM, emula architettura a 12 bit, in cui i registri sono differenti da architetture fisiche note. Questa VM implementa istruzioni macchina utili al resto del malware. Dentro la VM è incluso il malware, *scritto nel linguaggio macchina di questa architettura*. Anche se si ottenesse il malware, non esisterebbe un debugger per tale architettura. Ovviamente la *VM* può interagire col sistema *host*.

Nel livello esterno abbiamo misure per impedire l'apertura del file (anti-disassemblaggio, anti-vm, anti-debugger...).

Poichè deve esserci interazione con la macchina *host*, queste attività sono monitorabili con *analisi dinamica di base*.

Poichè il malware è numeroso e variegato, si introducono le *sandbox*.

#### Sandbox

Molte sandbox sono già esistenti, in cui le aree *sandbox* sono già disponibili, e richiedono in input il malware, da cui generano un report. Ne è un esempio il sito **Cuckoo**. Può capitare che il malware riconosca una sandbox già nota, e quindi alteri il comportamento. Possiamo costruire una *nostra* Sandbox, che il malware non conosce. Però non è scalabile, va bene per pochi malware.

#### Tecniche IA

Possiamo provare con l'intelligenza artificiale, ma non è una bacchetta magica, in quanto sono programmi che lavorano su particolari tipi di ingressi. Estrarre dal malware l'identificatore da fornire al programma di IA. Dimensioni delle sezioni etc, sono presi in modo *statico* o *dinamico* (se uso sandbox) e poi li passo al motore di IA. Ad oggi, un qualcosa del genere non è ancora pienamente realizzato. Inoltre, l'IA può essere usata per creare malware stesso. *Nè vincitori nè vinti, si esce sconfitti a metà....*

## Ritorno alla metodologia

Riprendiamo la trattazione riguardante la metodologia per l'analisi del malware. Questo perchè l'analisi di un malware può essere oggetto di cose giuridiche e di tribunali. Quindi il metodo con cui forniamo report deve essere affidabile. Le nostre motivazioni *non devono essere contestabili*, bisogna essere *rigorosi*. L'informatica è molto giovane, molto variabile, e quindi molti aspetti sono ancora oggetto di trattazione.

Una metodologia deve identificare una serie di fasi, per il lavoro di analisi. In particolare, possiamo trovare:

0. **Malware Injection**, il malware viene introdotto in qualche modo. L'analista non è l'attore di questa fase.
1) **Detection**, ci accorgiamo del contagio. Non è direttamente conseguente alla fase 1, può passare anche molto tempo! Qui inizia il vero lavoro.
2) **Isolation and Extraction**.
3) **Basic Static (code) Analysis**.
4) **Dynamic Static (behavior) Analysis**.
5) **Advanced Static(Code) & Dynamic (behavior) Analysis**, uso congiunto di *disassembler* e *debugger*.
    - 5.1) Se necessario, ritornare agli step 4 e 5.

6) **Patter Recognition**, identificazione di patter noti.

7) **Malware Inoculation & Remedy**, evitiamo che il malware continui a diffondersi.

Bisogna sempre partire da un **obiettivo (goal)**, avendo certi **dati (input)**, seguendo un **processo (process)** e producendo dei **risultati (output)**

**1 - Fase Detection**

E' consigliabile fare un'**analisi ambientale**. Supponiamo di avere disco o macchina accesa, la prima cosa da fare è copiare tutto l'hard disk. Abbiamo visto più volte che il malware si cancella, e comunque meglio non perdere informazioni del sistema.

- **Obiettivo**: il malware c'è, cerchiamo di identificare la sua natura, fornendo evidenze.
- **Input:** Sto ad inizio lavoro, non ho *output* precedenti. Immaginiamo che ci venga fornito il file, o una macchina che mal funziona.
- **Process**: Usiamo *antivirus* o *meta-engine*, per cercare evidenza malware. O guardiamo una parte specifica.
- **Output**: Numero di antivirus che riconosco il malware, o la categoria del malware. Qualcosa di tangibile ed usabile per le fasi successive.

**2 - Isolation and extraction**

Dopo aver identificato la fonte dell'attività, la estraiamo. Dobbiamo stare attenti a non auto-infettarci.

- **Obiettivo**: Estrarre il malware per portarlo sulla mia macchina di analisi.
- **Input:** Posizionamento del malware.
- **Process**: Si crea archivio protetto da password (anche banale), non bisogna correre il rischio di estrarlo ed eseguirlo per caso.
- **Output**: Archivio con *hash* crittografici, per avere la certezza che il file non sia stato alterato.

**3 - Basic Static (code) Analysis**

- **Obiettivo**: Rapidamente, collezionare informazioni sul malware.
- **Input:** Malware estratto dalla fase precedente.
- **Process**: Ricerca delle stringhe, tool per le testate *PE*, editor di risorse,...
- **Output**: Informazioni prodotte.

**4 - Basic Dynamic (behavior) Analysis**

- **Obiettivo**: Osservare, capire, documentare il comportamento del malware.
- **Input:** Malware, e hash calcolati nella fase precedente, usabili con sandbox.
- **Process**: Cuckos, CWSandbox, per eseguirci il malware, o usiamo la nostra VM (con *Process Monitor* o *regshot*.)
- **Output**: Informazioni prodotte sul comportamento del malware.

**5 - Advanced Static(Code) & Dynamic (behavior) Analysis**

- **Obiettivo**: Prelevare nuove informazioni, non ottenute dalle fasi precedenti.
- **Input:** Malware.
- **Process**: Disassembler, debugger.
- **Output**: nuove Informazioni prodotte sul comportamento del malware.

**6 - Pattern Recognition**

- **Obiettivo**: Avere visione generale: che faceva il malware? a chi serviva?
- **Input:** informazioni ottenute dalle fasi precedenti.
- **Process**: / (o il nostro cervello, in quanto produciamo noi la sintesi e correlazioni.)
- **Output**: descrizione della *big picture*, ovvero un resoconto su cosa faceva il malware, perchè lo ha fatto, come è successo.

**7 - Malware Inoculation & Remedy**

- **Obiettivo**: Prevenire e recuperare. Se il malware è *ransomware*, e quindi usava una chiave, allora dobbiamo ottenere quella chiave.
- **Input:** Tutte le info precedentemente ottenute.
- **Process**: Disassembler, debugger.
- **Output**: **IOC**, indicatori di compromissione, da dare in pasto ad **IDS** e **IPS**. Devono essere costruiti appositamente per questi sistemi. Se in futuro capita questo malware, sei addestrato per poterlo riconoscere o prevenire.