- MALWARE = qualunque software che causa danni all'utente e/o gira contro la volontà o all'insaputa dell'utente.
- SCOPO DELL'ANALISI DEL MALWARE (host-based signatures vs network signatures)

23/09/2021

RCE

RCE = Reverse Code Engineering

Definizione:

Il REVERSE ENGINEERING è il processo di estrazione della conoscenza e degli schemi di progetto di qualsiasi oggetto costruito dall'uomo.

Per certi versi è molto simile alla ricerca scientifica: consiste nell'estrarre da un insieme di dati quantitativi un modello formale che rappresenti in modo adeguato la "realtà", la quale non è direttamente desumibile dalla base di dati raccolti.

↳ Non è un lavoro meccanico: devono essere utilizzate opportune metodologie e l'intuizione personale.

Definizione:

Il REVERSE CODE ENGINEERING è il processo di ricostruzione delle finalità, delle strutture dati, dell'algoritmo e dei dettagli implementativi di un programma per un calcolatore elettronico.

TIPICAM. SI TRATTA DI UN FILE ESEGUIBILE CONTENENTE CODICE MACCHINA

L'input del RCE è una rappresentazione a basso livello di un programma.
L'output è una rappresentazione ad alto livello delle conoscenze apprese durante l'attività di reversing, come ad esempio:

→ La descrizione degli effetti del programma.
→ L'algoritmo del programma espresso in pseudo-codice.
→ La descrizione di un protocollo.

Motivazioni del RCE:

→ Produrre adeguata documentaz. del funzionamento di un programma sviluppa in passato. → LECITO

→ Sviluppare un programma che interagisca o utilizzi un altro programma di terze parti. → LECITO MA DIPENDENTEMENTE DALLE LICENZE DEL CODICE

→ Sviluppare un programma che si ponga come concorrente di un altro programma di terze parti. → MENO LECITO

→ Verificare che il funzionamento di un programma di terze parti non costituisca un rischio per la propria organizzazione. → LECITO MA VIENE VIETATO DALLE LICENZE D'USO DEI SISTEMI COMMERCIALI

→ Superare meccanismi di protezione di materiale digitale. → NON LECITO

→ Analizzare il comportamento del malware al fine di:

1) Determinare i suoi effetti.
2) Sviluppare un sistema di protezione.

} LECITO

3) Derivarne una nuova versione. → NON LECITO

→ Acquisire conoscenze sulle vulnerabilità dei sistemi informatici per attaccarli o per renderli più robusti.

→ Determinare debolezze teoriche o implementative di un'applicazione di crittografia.

L'attività di RCE non è illecita di per sé: lo diventa quando ha una finalità illegale oppure lede i diritti di qualcuno.

↳ Spesso non è semplice stabilire se un'attività di RCE è legale o meno.

▸ VEDERE DEFINIZIONE HACKER VS CRACKER

Il RCE può essere considerato come l'operazione inversa alla programmazione.

L'hacker deve seguire dei principi RCE per svolgere bene il suo lavoro di RCE.

Tali principi sono:

1) Maggiore è la comprensione dell'intero sistema, tanto più rapida, efficiente e produttiva è l'attività di reversing. → CONOSCENZE DI PROCESSORI, SISTEMI OPERATIVI, RETI DI CALCOLATORI, OPPURE I PRINCIPALI METODI DI OFFUSCAMENTO E RILEVAZI DI DEBUGGER E VM

2) Per capire il codice scritto da qualcun altro, è necessario capire come funziona il proprio codice. → NECESSITÀ DI SAPER PROGRAMMARE BENE IN MODO DA ESSERE IN GRADO DI RICONOSCERE GLI ALGORITMI CONCEPITI DAI PROGRAMMATORI

3) La qualità e la conoscenza degli strumenti di RCE determina la qualità dell'intero processo di reversing. → E.G. GHIDRA, OLLYDBG, ...

4) La chiave del reversing è l'abilità di identificare e comprendere gli schemi ricorrenti nel codice; di conseguenza, non esiste sostituto dell'esperienza!

Metodologia:

Quando si intraprenderà qualunque attività di RCE è necessario:
→ Avere ben chiari gli obiettivi che ci si prefigge.
→ Stabilire in che modo operare per ottenere gli obiettivi.
→ Verificare continuamente la correttezza delle informazioni ottenute.

Gli approcci fondamentali del RCE sono:
- Analisi black-box (o Live code analysis), che osserva il funzionamento del programma "dal vivo", eseguendolo (non sempre è fattibile).
- Analisi white-box (o Dead listing analysis), che analizza il codice macchina (può essere dispendioso).
- Analisi gray-box (o Mixed analysis), che osserva il funzionamento del programma eseguendo il suo codice macchina in modo controllato, e utilizza strumenti come debugger applicativi e di sistema (è efficace ma esistono tecniche ad-hoc per ostacolarla).

Di seguito viene proposto un esempio di metodologia proposto da Cesati:
1) DESCRIZIONE PRELIMINARE: si descrive ciò che si conosce del programma prima di iniziare il reversing.
2) FORMALIZZAZIONE DELL'OBIETTIVO: si stabilisce lo scopo con cui viene intrapresa l'attività di reversing.
3) OTTENIMENTO DEL CODICE MACCHINA: talvolta può essere necessario decriptare o de-offuscare il codice (questo in particolar modo quando il programma è impacchettato e/o cifrato).
4) OSSERVAZIONE DEL FUNZIONAMENTO: se possibile (e solo se è possibile), si adotta la tecnica di analisi black-box.

5) DISASSEMBLAGGIO DEL CODICE MACCHINA: è il passo che consente di cominciare l'analisi white-box, ed è un lavoro svolto da appositi disassemblatori. È in genere conveniente salvare il programma disassemblato in un documento che consenta di modificare il testo e di aggiungere commenti.

6) LOCALIZZAZIONE DI UN FRAMMENTO ASSEMBLY: se il codice macchina è lungo, è necessario identificare una porzione di codice che sia interessante per perseguire l'obiettivo del reversing.

7) ANALISI DEL FRAMMENTO ASSEMBLY: si analizza il frammento di programma in assembly per arrivare a comprendere le strutture dati utilizzate, le operazioni effettuate, le operazioni (funzioni) invocate, ecc. È essenziale annotare ogni informazione determinata in questa fase, anche se incompleta o ipotetica.

8) VERIFICA DEI RISULTATI: dopo aver raggiunto la finalità predefinita dell'attività di reversing, è essenziale cercare di verificare la correttezza delle conclusioni raggiunte con una procedura possibilmente differente da quella impiegata per il reversing.

9) RIEPILOGO DELLE INFORMAZIONI OTTENUTE: alla fine è necessario riepilogare in un documento scritto tutto il processo di reversing, i risultati ottenuti e le informazioni collaterali apprese durante il processo.

28/09/2021

*QUESTO PER NON ESSERE ATTACCATI

Il 98% del malware attacca un sistema Windows ⟹ affinché il malware non venga eseguito sulla nostra macchina, è può essere opportuno utilizzare un sistema Linux.*
D'altra parte, può essere utile eseguire il codice in maniera controllata. Per questo motivo, è comodo avere una macchina virtuale (con una versione appropriata di Windows).
→ MAI far girare il Malware sulla macchina host (→ solo sulla macchina virtuale)!

Il nostro obiettivo è effettuare il RCE:

codice macchina → RCE → info ad alto livello

LIVELLI DI ASTRAZ. DEI PROGRAMMI:
Hardware - microcode - machine code - low level language - high level language - interpreted language

ARCHITETTURA VON NEUMANN: CPU + RAM + I/O

Unità di compilazione

SNACCHERO PARSER

Contiene le istruzioni macchina relative al file sorgente

File oggetto 2

Libreria di sistema 1

Libreria di sistema 2

File sorgente → [Pre compilatore] → [Compilatore] → File oggetto → [Linker] → File eseguibile

- Include dei file quando incontra #include <nome.h>
- Include un'estensione quando incontra #define

La differenza tra file oggetto e file eseguibile sta nei riferimenti (è il linker che mette a posto i riferimenti anche tra file oggetto differenti). Il linker effettua collegam. anche tra file oggetto e librerie di sistema.

→ PROCESSO ≡ istanza di un programma in esecuzione (dove programma ≡ insieme di istruz.)

↳ Il codice linkato staticam. contiene tante istruzioni macchina che sono + difficili da interpretare. (*)

↳ Il codice linkato dinamicamente è + breve e + facilmente interpretabile (ma si può mascondere / camuffare + facilmente...).

↳ Posso anche avere codice linkato a runtime, tramite l'utilizzo delle dll LoadLibrary, GetProcAddress.

Alla fine arriviamo a un codice macchina prodotto. (DOPO ESSERE PASSATI PER PRE-COMPILATORE, COMPILATORE E LINKER)

Il lavoro di decodifica di un codice macchina è fatto da un DISASSEMBLATORE.

DISASSEMBLATORE
- → LINEARE → e.g. OBJDUMP
- → INTERATTIVO (necessario per lavori + complicati) → e.g. IDA PRO – GHIDRA

→ ARCHITETTURA VON NEUMANN (utilizzata da noi): i programmi e i dati sono salvati insieme in memoria.

⇒ Se nel file eseguibile troviamo '103', non sappiamo a priori se si tratta di un'istruz. macchina, di un carattere ASCII, di un valore numerico, ecc.: il disassemblatore dovrebbe "tirare a indovinare". Per questa cosa, il disassemblatore lineare non basta.

Un disassemblatore interattivo richiede l'interazione con l'utente: magari è quest'ultimo che indica al disassemblatore che una certa cosa è una porzione di codice e non altro.

Il file oggetto è suddiviso in sezioni:

→ TEXT (codice macchina)

→ DATA (variabili inizializzate)

(*)
- LIBRERIA STATICA → quando il linker va a costruire il file eseguibile ogni volta che gli serve una funzione (e.g. printf()), va a cercarsi in un apposito file il codice macchina che realizza la printf() e lo copia all'interno dell'eseguibile: a parte così.
- LIBRERIA DINAMICA → il linker registra dentro al file eseguibile l'informazione che una certa funzione di quel programma sarà fornita a run-time dal sistema operativo.

LIBRERIA DINAMICA
- → Windows: DLL
- → Linux: SO (Shared Object)

→ BSS (~~...~~ descrizione di variabili di fatto non inizializzate che dovranno essere inizializzate a zero quando il programma verrà eseguito).

→ DATI READ ONLY (costanti, stringhe esplicite "come questa").

Nel file eseguibile, invece, + sezioni sono collassate in segmenti: ad esempio, in un certo segmento potrebbero andare a finire sia istruzioni macchina, sia dati read only.

D'altra ~~parte~~ parte, abbiamo il processo, che utilizza la memoria suddivisa in PAGINE, ciascuna delle quali corrisponde a un segmento del file eseguibile.

→ FORMATO DEL FILE ESEGUIBILE DI LINUX: ELF → Executable and Linkable Format

→ FORMATO DEL FILE ESEGUIBILE DI WINDOWS: PE-COFF → PE (32 bit) ← processore a 32 bit; PE+ (64 bit) ← processore a 64 bit

Portable Executable; Common Object File Format

→ Abbiamo 2 Instruction Set (ISA) differenti: ~~a~~ istruz. macchina a 32 bit e istruz. macchina a 64 bit. Questo perché i processori moderni (a 64 bit) supportano anche programmi a 32 bit (detti x86; i programmi a 64 bit sono detti x64).

↳ Per questo motivo, esiste ancora ~~un~~ malware scritto a 32 bit in modo tale che possa intaccare + macchine possibili. Ciò non toglie che ∃ malware scritto a 64 bit.

I processori INTEL sono LITTLE-ENDIAN (LE). → In rete, globalmente, si utilizza invece il BIG ENDIAN (BE).

- byte: 8 bit
- word: 4 byte
- short: 2 byte
- quadword: 8 byte

LITTLE-ENDIAN vuol dire: all'indirizzo di memoria + piccolo metto il byte meno significativo.

↳ ESEMPIO: $515 = \underset{\text{BYTE + SIGNIFICATIVO}}{2} \times 256 + \underset{\text{BYTE - SIGNIFICATIVO}}{3}$ → in little-endian è scritto come 3, 2.

INDIRIZZO DI MEMORIA + PICCOLO

Ghidra usa la sintassi assembler dell'Intel. → METTE L'OPERANDO DESTINAZ. PER PRIMO E L'OPERANDO SORGENTE PER SECONDO NELLE OPERAZIONI

Objdump usa la sintassi assembler dell'AT&T → L'OPPOSTO DI AT&T

20/09/2021

ENTRY = posizione dentro al file eseguibile, ma anche posizione in memoria in cui inizia l'esecuzione del programma.

→ Quando facciamo RCE, abbiamo due visioni:

FILE ESEGUIBILE	MEMORIA (RAM)
→ Segmenti: • TEXT, • DATA, • BSS, • RDATA (READ ONLY)	• CODICE, • DATA
	• HEAP → Allocaz. dinamica della memoria
	• STACK → Serve per l'allocaz. delle funzioni e per la dichiarazione delle variabili automatiche

VARIABILI:
- GLOBALI → in data → VISIBILITÀ: tutto il programma; PERSISTENZA: totale (finché il programma gira)
- LOCALI (= automatiche) → nello stack → VISIBILITÀ e PERSISTENZA: funzione in cui sono definite
- STATIC GLOBALI → in data → VISIBILITÀ: unità di compilazione (ed eventuali file sorgente); PERSISTENZA: totale
- STATIC LOCALI → in data → VISIBILITÀ: funz. in cui sono definite; PERSISTENZA: totale

FILE ESEGUIBILE	MEMORIA (RAM)
NOME SIMBOLICO: l: mov eax, 1	
Utilizzo di indirizzi logici	Utilizzo di indirizzi fisici

Tiene dunque traccia dell'indirizzo del segmento CODE

- REGISTRO CS: assegna offset di indirizzi logici rispetto all'inizio del segmento CODICE.
- REGISTRO DS: " " " " DATA.
- REGISTRO SS: " " " " STACK.

In una memoria piatta, lo spazio di indirizzi del codice, quello dei dati e quello dello stack coincidono. → Significa che CS=DS=SS=0

INDIRIZZO LOGICO —segmentaz.→ INDIRIZZO LINEARE —paginaz.→ INDIRIZZO FISICO

EFFETTUATA DALL'UNITÀ DI SEGMENTAZIONE

EFFETTUATA DALL'UNITÀ DI PAGINAZIONE CON L'USO DELLE APPOSITE TABELLE (ad tabella di pagina/tavola)

INDIRIZZO LINEARE = base del segmento all'interno dell'address space + offset all'interno del segmento (non è altro che un posiz. effettiva all'interno del contenitore)

Si parla di INDIRIZZO VIRTUALE quando il mapping tra indirizzo lineare e indirizzo fisico non è fisso.

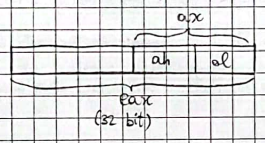
Registri del processore Intel (32 bit):

→ eax con le sue sottoparti ax | ah | al
→ ebx " bx | bh | bl
→ ecx " cx | ch | cl
→ edx " dx | dh | dl
→ esi " si
→ edi " di
→ ebp " bp

Utilizzati dalla CPU durante l'esecuzione di un programma tipicam. x memorizzare dati o indirizzi

REGISTRI GENERAL PURPOSE

Puntatore a una certa variabile/dato all'interno dello stack; a differenza di esp, è fisso, per cui consente facilmente di tenere un riferimento stabile rispetto alle variabili all'interno dello stack. *



* Più precisamente, in genere, rimane fisso nell'ambito dell'esecuzione di una singola funzione, in modo tale da tenere traccia di qual è la zona dello stack propria della funzione stessa.
↳ IN PARTICOLARE QUALI SONO I PARAMETRI E LE VARIABILI LOCALI

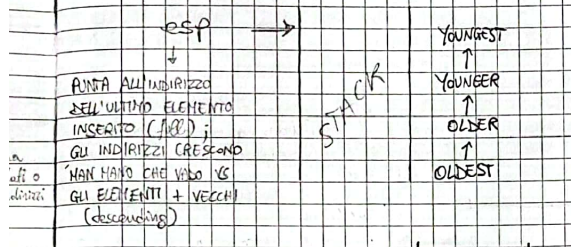
→ REGISTRI SEGMENTO: cs, ds, es, fs, gs, ss → Usati per tenere traccia delle sezioni in memoria
→ esp (stack pointer) → il libro lo mette tra i registri general purpose
→ eflags (registro dei bit di stato) → Usato per prendere decisioni; contiene 32 flag
→ eip (instruction pointer → memorizza l'istruzione macchina da eseguire)
" program counter
→ cr-

→ mmx0, ..., mmx7
→ fpr0, ..., fpr7
→ xmm0, ..., xmm15

REGISTRI DEL CO-PROCESSORE

PER QUANTO RIGUARDA IL PROCESSORE A 64 BIT..
→ r8, ..., r15
→ I registri che iniziavano con e- nel processore a 32 bit, hanno una r al posto della e.

Lo stack del processore Intel è "FULL DESCENDING", in cui l'unità base in cui lavoriamo è 4 byte. Lo schema è:



→ FUNZIONE CALL: invocazione di una funzione f, salvataggio dell'indirizzo h (relativo al comando successivo alla CALL) sullo stack, eip ← f. esp ← h

→ FUNZIONE RET: terminazione della funzione f, pop dell'indirizzo h dallo stack, eip ← h.

Supponiamo di avere la seguente funzione con parametri:

f(int n, int m, int q)

Abbiamo 2 convenzioni per il passaggio di parametri:

• CDECL

32 BIT	64 BIT
→ Passaggio dei parametri sullo stack in ordine inverso (push q - push m - push n - call f)	
→ La pop dei parametri viene effettuata dal chiamante	
eliminaz. di 3 elementi dallo stack ← (add esp, 0xc) 12	

*PIÙ IN DETTAGLIO:
1) Push dei parametri sullo stack
2) Chiamata della funzione (⇒ push di EIP su stack)
3) Prologo (⇒ allocaz. spazio x variabili locali + push di EBP sullo stack)
4) Esecuzione del corpo centrale della funzione
5) Epilogo (⇒ ripristino di stack e registro EBP)
6) Chiamata a ret (⇒ pop di EIP dallo stack)
7) Pop dei parametri dallo stack

• STDCALL → La differenza da CDECL sta nel fatto che la pop dei parametri viene effettuata non dal chiamante, bensì dal chiamato (che nel nostro esempio invoca una ret 12). → eliminaz. di 3 elementi dallo stack

↳ QUESTA È LA CONVENZIONE STANDARD X LE WIN-API

• FASTCALL → La differenza da CDECL sta nel fatto che i primi parametri (tipicam. 2) non vengono posizionati sullo stack, bensì vengono passati alla funzione chiamata tramite l'utilizzo di registri (tipicam. EDX-ECX).

È un'area di memoria apposita per le funzioni, ma contiene anche variabili locali e contesto di flusso.

05/10/2021

Configurazione dello stack dopo la chiamata della funzione f da parte della funzione h:

var 2
var 1
h (= indirizzo ritorno)
arg 1
arg 2
arg 3

PARTE DELLO STACK TIPICAM. LASCIATA LIBERA NEL PROLOGO TRAMITE UN'ISTRUZIONE SUB

Qui in mezzo in realtà ci viene messo old EBP, che viene puntato dal registro EBP per tenere traccia di variabili, argomenti, ecc. *

In particolare old EBP serve a ripristinare correttam. il registro EBP quando il controllo torna al chiamante.

* EBP permette di accedere a parametri e variabili locali in modo agevole, semplicemente utilizzando uno spiazzamento rispetto a EBP stesso.
↳ EBP = puntatore allo stack frame della funzione

- AND ESP, 0xfffffff0 → azzeram. degli ultimi 4 bit dell'indirizzo di ESP
  ⇒ allineamento di ESP a 4×4 = 16 byte.

```
PUSH EBP
MOV  EBP, ESP
SUB  ESP, n
```

} Questo insieme di istruzioni è (quasi) equivalente all'istruzione ENTER n, m

↳ Annidamento della funzione (in C, C++ vale zero, perché sono linguaggi che non prevedono annidamenti di funzioni)

→ LEAVE è l'opposto di ENTER ed è equivalente a:

```
MOV ESP, EBP
POP EBP
```

→ ripista lo stack esattam. com'era all'inizio della chiamata alla funzione

ENTER = prologo della funzione

LEAVE = (+ RET) = epilogo della funzione

In fase di compilazione, è possibile omettere l'utilizzo del frame pointer (l'utilizzo del registro EBP) per motivi di efficienza: in INTEL X86 ci sono pochi registri, e si può lasciare EBP per altri scopi (⇒ per recuperare variabili/argomenti delle funzioni, in tal caso, bisogna rifarsi sempre tutti i calcoli).

NB: entry ≠ main ! Prima del main, bisogna eseguire del codice atto a preparare l'ambiente di esecuzione.

EBX, ESI, EDI, oltre a EBP, sono i registri da preservare nella chiamata di una funzione: vengono inseriti nello stack nel prologo della funzione, e vengono ripristinati tramite una POP nell'epilogo.

Istruzione LEA = Load Effective Address → serve per registrare l'indirizzo di un elemento o anche un qualunque valore.

07/10/2021

Istruzione SAR = Shift Arithmetic Right → è una SHIFT RIGHT che preserva il bit del segno in prima posizione.

ESEMPIO: 1abcdefg $\xrightarrow{SAR \gg 1}$ 11abcdef

Istruzione CDQ → estende a 64 bit il contenuto di un registro a 32 bit. Più precisamente, estende EAX nella coppia di registri EDX-EAX, copiando il bit del segno di EAX in tutte e 32 le posizioni di EDX.

Istruzione ADC → somma i due operandi, aggiungendoci anche il CARRY FLAG.

Il costrutto switch in Assembly può essere tradotto con una JUMP TABLE.

→ Il compilatore può decidere di mettere, al posto della chiamata a una funzione (istruzione CALL), il corpo della funzione stessa in Assembly: lo fa per motivi di ottimizzazione.

EAX è il registro in cui viene memorizzato il valore restituito da una funzione.

12/10/2021

RCE di linguaggi orientati agli oggetti:

JAVA: il BYTE CODE ha una struttura molto simile al linguaggio ~~orientato~~ Java stesso → a meno di offuscamenti, dovrebbe essere non troppo complicato effettuare RCE con Java.

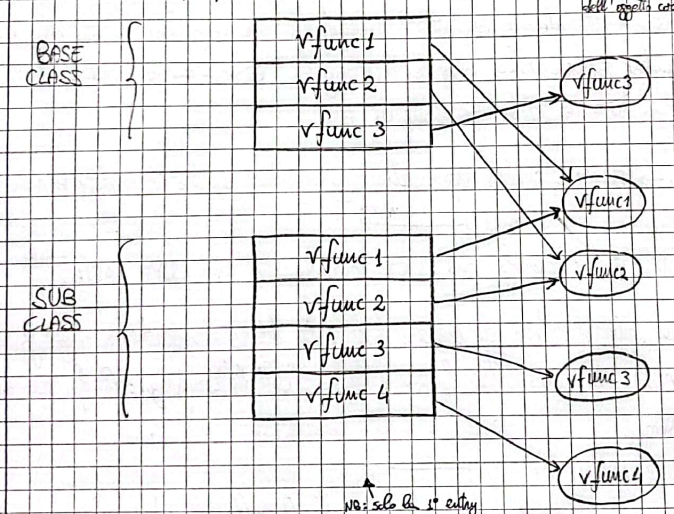
In ogni caso, molto malware è scritto a oggetti: in Windows si utilizza Visual Studio con C++ a tal proposito. C++ è in certi versi + sofisticato di C, per cui il suo codice macchina è (molto) + complicato. In particolare, la complicazione è dovuta a polimorfismo, ereditarietà, overloading, e altri meccanismi propri dei linguaggi O.O.

↳ CLASSE IN C++ = particolare struttura C che ha un puntatore agli indirizzi dei metodi. (*)

↳ Effettivam., la programmazione a oggetti può essere fatta anche con linguaggi NON orientati a oggetti: osserviamo che il codice macchina e il codice assembly derivanti da C++ non hanno nulla a che fare con la programmaz. a oggetti così come la conosciamo.

(*) Questi indirizzi di metodi, se si tratta di metodi VIRTUALI, si trovano in una vtable:

→ La vtable (= Virtual function table) non è altro che un array di puntatori a funzioni che viene referenziato dai primi 4 byte dell'oggetto corrispondente.



LE CHIAMATE AI METODI NON VIRTUALI VENGONO RISOLTE A TEMPO DI COMPILAZIONE, PER CUI SI PRESENTANO COME DEL LE SEMPLICISSIME
call address_fun
LE CHIAMATE AI METODI VIRTUALI, INVECE, ESSENDO RISOLTE A RUNTIME, SI PRESENTANO COME
call eax
DOVE EAX CONTIENE L'INDIRIZZO DELLA ENTRY DELLA VTABLE DOV'È MEMORIZZATO L'INDIRIZZO DEL METODO CHE DEV'ESSERE INVOCATO.

In questo specifico caso la sub-class ridefinisce vfunc3.

NB: solo la 1ª entry della vtable viene acceduta direttam. (le altre solo con spiazzamento).

↑ Implementaz. (ovvero codice) dei metodi

In C++ i metodi hanno un parametro implicito (è il primo): this, che è un puntatore all'istanza della classe con cui stiamo lavorando.

↳ In Microsoft Visual C tale parametro NON viene salvato sullo stack, bensì nel registro ECX.

↳ QUESTA CONVENZIONE DI CHIAMATE A FUNZIONI È NOTA COME thiscall.

14/10/2021 — Applicazioni Windows: (msdn)

La funzione iniziale del programmatore nelle applicazioni GUI Windows ~~sono~~ è detta WINMAIN (o wWINMAIN). **

x USO STRINGHE ASCII — x USO STRINGHE UNICODE

→ La Microsoft non documenta tutti gli aspetti delle ~~dell~~ applicazioni Windows: chi scrive malware sfrutta questo per "nascondersi".

** int WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, INT nShowCmd)

- HINSTANCE: TIPO DI DATO CHE INDICA UN RIFERIMENTO (non x forza INDIRIZZO) → HANDLE = identificatore opaco
- hInstance: ID processo
- hPrevInstance: oggi è sempre 0
- lpCmdLine: linea di comando in cui è stata effettuata la chiamata (argomenti)
- nShowCmd: come deve essere aperta la finestra (minimizzata, schermo intero, ...)

Le app Windows, oltre ai dati e al codice, utilizzano le risorse (e.g. ICONE).

→ HICON LoadIconA (HINSTANCE hInstance, LPCSTR lpIconName)

- hInstance: handle dell'istanza che contiene la risorsa
- lpIconName: nome della risorsa

↳ QUESTA FUNZIONE SERVE A CARICARE UN'ICONA

→ ATOM RegisterClassEx (const WNDCLASSEXA nameParam)

↳ QUESTA FUNZIONE REGISTRA UNA CLASSE FINESTRA NELL'APPLICAZIONE

→ WNDCLASSEXA è una struttura.

Message loop (con GetMessage(), TranslateMessage(), DispatchMessage()) → ciclo while che tiene vita alla finestra finché l'applicazione non termina.

Questo loop si trova nel WinMain.

- GetMessage() → RICEZIONE DI UN MESSAGGIO-EVENTO PROVENIENTE DAL S.O. WINDOWS.
- TranslateMessage() → TRASFORMAZ. DEL MESSAGGIO PRIMA CHE VENGA GESTITO.
- DispatchMessage() → GESTIONE DEL MESSAGGIO ~~E RESTITUZIONE DEL CONTROLLO AL S.O.~~

  Viene effettuata dalla WINDPROC (che è una funz): il messaggio viene reinviato al S.O. e è quest'ultimo che invoca la WINDPROC con questo messaggio

  ↳ defWindProc() → gestisce il messaggio nella modalità di default.

26/10/2021

L'analisi statica del codice è molto laboriosa. Esistono delle scorciatoie.
In particolare, quando effettuiamo reversing, possiamo avere:

→ ANALISI STATICA
- DI BASE
- AVANZATA → Noi finora ci siamo occupati di questo (DISASSEMBLAGGIO)

→ ANALISI DINAMICA
- DI BASE
- AVANZATA

• Analisi statica di base = analisi (statica) di un eseguibile senza effettuare disassemblaggio.
L'eseguibile (in formato PE) è composto da tante sezioni:

HEADER
TEXT
DATA
DATA READ ONLY (RDATA)
RISORSE

← L'header contiene i metadati del file, come gli import, le dimensioni (VIRTUAL SIZE = spazio allocato per una sezione durante il caricamento del processo; SIZE OF RAW DATA = dimensione della sezione su disco), il timestamp di compilazione e le info su se si tratta di un'applicazione GUI oppure command-line.

A volte abbiamo anche le sezioni IDATA - EDATA, che contengono info sulle funzioni rispettivamente importate ed esportate. Se tali sezioni non sono presenti, le info si trovano in RDATA. →

← Alla fine c'è anche la sezione RELOC, che contiene le info per la rilocazione dei file di libreria.

L'analisi statica EVITA le sezioni text e data (perché richiedono disassemblaggio).
Comunque, può fornire delle informazioni utili.

• Analisi dinamica di base = analisi che consiste nel "dare vita" a un programma e nel capire cosa fa a run-time (e.g. quali sono i file che vengono aperti/chiusi/modificati? Cosa fa il programma nella rete?)
Ciò non richiede l'analisi del codice: fare tutto questo sul malware richiede MOLTA attenzione.

• Analisi dinamica avanzata = analisi che fa uso del DEBUGGER, che consente di coordinare l'analisi statica del codice con la possibilità di eseguirlo.

Gli approcci al reversing sono:

→ "WHITE BOX" { BASIC STATIC ANALYSIS / ADVANCED STATIC ANALYSIS

→ "BLACK BOX" { BASIC DYNAMIC ANALYSIS

→ "GRAY BOX" { ADVANCED DYNAMIC ANALYSIS

RIASSUMENDO:
→ Hashing
→ VirusTotal
→ Strings
→ PE view
→ PE Lear
→ Load PE
→ PE Studio
→ Resource Hacker
} ANALISI DELLA TESTATA

IL LIBRO NOMINA ANCHE:
→ PE explorer (ANALISI DELLA TESTATA)
→ dependency Walker (ANALISI DELLE FUNZIONI LINKATE DINAMICAM.)

Basic static analysis:

Tool utilizzati:

- HASHING → calcolo dell'hash dell'eseguibile (e.g. con MD5, SHA256).

VirusTotal è un sito che fa da collettore di antivirus e di malware. Se gli diamo in pasto un programma, lo analizza e si salva l'hash coi risultati di tutti gli antivirus. Se qualcun altro gli dà in pasto lo stesso programma, VirusTotal non deve rifare l'analisi.

- RICERCA DELLE STRINGHE ALL'INTERNO DELL'ESEGUIBILE → comando strings nome_eseguibile in Linux.

Questo tool restituisce (di default) tutte le stringhe di almeno 4 caratteri presenti nel programma.

↳ strings -n 2 nome_eseguibile: restituisce tutte le stringhe di esattam. 2 caratteri.

Su Windows, questa stessa funzionalità ce l'ha un tool chiamato Strings.
(usare sempre il terminale a tal proposito).

⇒ Nascondere le stringhe è la 1° cosa che chi scrive malware fa (e.g. salvando le stringhe in un formato codificato). → Perché effettivam. le stringhe rilasciano parecchia informazione.

Le stringhe potrebbero non vedersi anche perché l'eseguibile può essere stato compattato e/o offuscato.

Compattare un programma con un packer può scoraggiare l'analisi white box e gray box ⇝ chi analizza l'eseguibile, in tal caso, deve prima de-compattare il prog.

⇝ Le stringhe fanno parte della sezione data read only.

- ANALISI DELLA TESTATA (HEADER) DEL PROGRAMMA

Un tool su Windows che fa quest'analisi è PE view. → PE view offre informazioni come VIRTUAL SIZE, SIZE OF RAW DATA e le RVA

RVA = indirizzo virtuale relativo (è un offset rispetto all'inizio di una specifica sezione).

Poi abbiamo anche PE-Bear, che in + permette di collegare un elemento con la sua posizione (byte della) all'interno del file. Questo è molto utile quando si vuole modificare qualche testata del file.
Un terzo programma è Lord PE, che è disponibile anche su Linux.
Un programma un po' più sofisticato è PE Studio, che, oltre all'analisi delle testate, fa qualcosa in +: calcola le firme, fornisce la data di compilazione, ecc.
~~Poi abbiamo Resource Hacker, che analizza le risorse all'interno di un file.~~

- ANALISI DELLE RISORSE DEL PROGRAMMA → ottenimento delle immagini utilizzate nel programma, delle icone, dei menù, delle finestre di dialogo, ecc.
  Resource Hacker è un tool che effettua questo tipo di analisi.
  ⇝ Molte volte, le stringhe sono classificate come risorse.

∃ un tool, PEiD, che, oltre a effettuare analisi statica, ha un punto forte: i plugin, che, di base, eseguono parti di programma (QUINDI ATTENZIONE!)
⇒ PEiD è già uno strumento di analisi dinamica.
Questo tool è potente perché è anche in grado di riconoscere programmi offuscati, di trovare il VERO entry point dei programmi e di trovare delle costanti cifrate all'interno degli eseguibili (con le relative funzioni crittografiche).

28/10/2021

<u>Basic dynamic analysis:</u>

SANDBOX = meccanismo per l'esecuzione di programmi "untrusted" in un ambiente virtuale safe; effettua in modo automatico analisi di base, producendo alla fine un documento pdf riepilogativo.

Tool utilizzati:

- PROCESS EXPLORER → mostra i processi attivi all'interno del sistema e, di ciascun processo, è in grado di descrivere le proprietà dell'eseguibile. Inoltre, può effettuare anche il dump della memoria. → Permette anche di capire se un documento (pdf/word, ..) lavora un malware o meno.
  ↳ Quest'ultima funzionalità è molto importante nel momento in cui abbiamo file eseguibili offuscati.

→ Inoltre può effettuare un confronto tra le stringhe dell'eseguibile su disco e le stringhe dell'eseguibile in memoria RAM.
↳ Permette anche di eseguire delle azioni sui processi (come sospenderli, terminarli, ecc.).

In effetti, un file eseguibile offuscato è composto da una porzione di codice in chiaro (che ha lo scopo di deoffuscare la porzione successiva) e una porzione di codice offuscato (quello vero e proprio). Nel migliore dei casi, quando un processo viene caricato in memoria, lì possiamo trovare tutto il codice in chiaro dell'eseguibile.

↳ Altro caso interessante: processo che non esegue il codice del file eseguibile con cui è partito.

- PROCESS MONITOR → registra tutto ciò che accade nel sistema. (≡ registra tutti gli eventi).
- REGSHOT → indica come il registro di sistema (Windows Registry) è cambiato tra un certo istante e un'altro.
- WIRESHARK → sniffa i pacchetti di rete.
- FAKE DNS → fa in modo che una richiesta destinata a un qualche server remoto venga in realtà soddisfatta da un host che ha l'indirizzo IP che gli diciamo noi.**

  Essa viene intercettata per fare di entità al DNS del SO di default.

  **QUESTO VIENE FATTO TRAMITE UN SERVER DNS LOCALHOST (PORTA 53)
- INETSIM (PER LINUX) → consente di realizzare decine di servizi web fake. Serve per costruire un ambiente in cui il malware può ricevere delle risposte alle richieste che lui fa (delle risposte fake chiaramente*); in questo modo, il malware non ottiene quello che si aspetta ma comunque non si blocca a causa di risposte di tipo "404 not found". → MOLTO SPESSO INETSIM E FAKEDNS COLLABORANO.

  *ma tecnicam. corrette; tra l'altro sono customizzabili.
- NETCAT → acquisisce informazioni sulle connessioni di rete + simula connessioni Internet.

Analisi del malware:

Si fa su una macchina virtuale; prima di caricarci un qualunque malware, è necessario salvare uno snapshot della macchina per salvare il suo stato che verrà recuperato (o alla fine dell'analisi) nel caso in cui il malware dovesse far danni. Dopodiché è necessario disabilitare la protezione che offre Windows contro virus e malware.

NB: Non è detto che il malware funzioni sulla versione di SO che abbiamo come macchina virtuale ⇒ può essere necessario avere molteplici macchine virtuali con versioni di SO diverse.

È buona norma lavorare scollegati dalla rete e aver disabilitato le cartelle condivise con la macchina host e altre eventuali macchine virtuali.

02/11/2021

Advanced dynamic analysis:

Si basa sull'utilizzo dei DEBUGGER.

Il debugger è uno strumento che serve per analizzare il codice e per verificare se il comportamento del programma corrisponde alle aspettative del programmatore (aiuta a effettuare il debug). In pratica, esegue codice in maniera controllata.

I debugger possono essere SOURCE LEVEL o ASSEMBLER LEVEL.

LAVORANO AD ALTO LIVELLO, SUL CODICE SORGENTE

LAVORANO A LIVELLO DEL ISTRUZ. MACCHINA: SONO QUELLI CHE UTILIZZEREMO

Inoltre, possono anche ~~essere~~ lavorare in KERNEL MODE o in USER MODE.

MODALITÀ NON PRIVILEGIATA

Tipicamente, per eseguire il debugger sul Kernel devo usare MODALITÀ PRIVILEGIATA

* necessario: istruzioni e macchine diverse. *

Una terza distinzione è tra debugger che lavorano in MODALITÀ REMOTA e debugger che lavorano in MODALITÀ LOCALE.

IL PROGRAMMA DA ANALIZZARE SI TROVA SU UNA MACCHINA DIFFERENTE

Infine, esistono i debugger GUI-BASED e i debugger COMMAND LINE-BASED.

Vediamo qualche esempio di debugger.

GHIDRA:

- È un meta-debugger: si interfaccia con dei debugger a basso ~~basso~~ livello (ed eseguono loro il lavoro).
- Ha un database (detto TRACCIA) in cui tiene le informazioni acquisite tramite il debugging (le informazioni sull'esecuzione). Da tali informazioni, è possibile ripercorrere l'esecuzione (la traccia) sia in avanti che all'indietro.
- È primitivo: permette di fare solo poche cose essenziali.
- Non è molto facile da usare.
- Lavora sia in modalità remota che in modalità locale.
- È assembler level.

* Infatti, una volta che il Kernel è ad esempio bloccato da un breakpoint, nessun processo può funzionare all'interno del sistema.

- Lavora sia in user mode che in Kernel mode.
- I debugger a basso livello con cui si interfaccia sono GDB (PER ANDROID, LINUX, WINDOWS...) e WinDbg (SOLO PER WINDOWS).

IDAPRO:
- È assembler level.
- Lavora principalmente in user mode.
- Lavora sia in modalità remota che in modalità locale.

GDB (GNU debugger):
- È assembler level.
- Lavora principalmente in user mode.
- A differenza dei due precedenti, è command line-based.
- Lavora sia in modalità remota che in modalità locale.

WINDBG:
- È il debugger ufficiale di Microsoft.
- Ha un paio di interfacce grafiche (WinDbg, WinPreview); tuttavia, i motori veri e propri ricevono solo comandi command-line.
- Lavora sia in user mode che in Kernel mode.
- Se in user mode, può lavorare sia in locale che in remoto; se in Kernel mode, può lavorare solo in remoto.
- I suoi componenti si trovano all'interno del SDK di Windows.

SOFTWARE DEVELOPMENT KIT DELLA MICROSOFT CHE CONTIENE DOCUMENTAZIONE, INTESTAZIONI, LIBRERIE, METODI E STRUMENTI PER LA CREAZIONE DI APPLICAZIONI WINDOWS

SOFTICE:
- È assembler level.
- Lavora in Kernel mode ma in locale (cosa impossibile per WinDbg).
- È stato dismesso: il suo ultimo aggiornamento c'è stato nel 2007, per cui non funziona nei sistemi operativi usciti dopo il 2007.

OLLY DBG:

- È il debugger che useremo noi, perché è utilizzatissimo per il debugging del codice a 32 bit; tuttavia, non supporta codice a 64 bit.
- È assembler level.
- Lavora in user mode.
- Lavora solo in modalità locale.
- È GUI-based.
- Noi usiamo la versione 1 di Olly Dbg, ma esiste anche una versione 2, che è completamente differente; dovrebbe essere una miglioria della versione 1 ma dagli ana listi è ancora considerata una versione beta.
- Da Olly Dbg deriva Imm Dbg, in cui è possibile estendere la funzionalità tramite Python.
  - È una serie di versione

$\exists$ un bit (TRAP FLAG - TF) che per ogni processo che, se impostato a 1, fa sì che, all'esecuzione di ogni istruzione, venga generata una DEBUG EXCEPTION.
- Si trova nel flag register
- ↳ Questo è il meccanismo usato dal comando STEP INTO di Olly Dbg.

04/11/2021

Breakpoint:

$\exists$ due tipi di breakpoint:

→ DI ESECUZIONE: si verifica quando il processore tenta di eseguire l'istruzione macchina che si trova in un determinato indirizzo.

→ DI ACCESSO: si verifica quando il processore tenta di effettuare un'accesso ai dati in memoria.

Una seconda classificazione di breakpoint è:

→ HARDWARE: consiste nel memorizzare un breakpoint address che, a ogni istruzione, verrà confrontato con l'instruction pointer da un componente hardware.

→ SOFTWARE

È chiaro che, se l'eseguibile è auto-modificante, si potrebbe perdere traccia dei software breakpoint.
- Questo è un caso di software breakpoint

▸ Istruzione macchina INT3: se eseguita, solleva una debug exception (è un modo alternativo
- VIENE MESSA DAL DEBUGGER AL POSTO DEL 1° BYTE (DEI PRIMI 2 BYTE) DELL'ISTRUZIONE A CUI CI VOGLIAMO FERMARE
- In quella versione a 1 byte, l'istruzione INT3 corrisponde al byte CC.

Questo è l'altro caso di software breakpoint

il settaggio del trap flag per segnalare la presenza di un breakpoint software di esecuzione).

CONTROMISURE DEL MALWARE CONTRO I BREAKPOINT:
→ Controllare il valore del trap flag.
→ Controllare la presenza di istruzioni INT3.

* Questo perché, di fatto, ∃ 4 registri speciali che tengono traccia degli hardware breakpoint.
Il malware potrebbe prendere delle contromisure anche contro questi 4 registri.

HARDWARE BREAKPOINT DI ACCESSO: hanno un indirizzo base e una dimensione. Hanno una grande limitazione: ne possono essere usati al più 4 (compresi gli hardware breakpoint di esecuzione).*

SOFTWARE BREAKPOINT DI ACCESSO: sono basati sulla protezione delle pagine in memoria.
Chiaram. si riferisce a 1 qualche locazione in memoria.
↳ MODIFICA DEL FLAG D'ACCESSO.

BREAKPOINT CONDIZIONALI: → SONO SOFTWARE BREAKPOINT
possono catturati o gestiti dal debugger
Sono in ogni caso attivi, ma vanno a verificare una particolare condizione per stabilire se intervenire x interrompere il flusso di esecuzione dell'app.

Eccezioni:

Quando viene lanciata un'eccezione, può essere gestita da:
→ Una procedura presente nell'app.
→ Una routine del SO predefinita.
→ Il debugger (first-chance). → IL DEBUGGER HA UNA 1ª POSSIBILITÀ DI GESTIRE UN'ECCEZIONE
↳ C'è anche una second-chance → SE IL DEBUGGER, ALLA 1ª CHANCE, DECIDE DI FAR GESTIRE LE ECCEZIONI AL PROGRAMMA E, DURANTE LA GESTIONE, VIENE SOLLEVATA UNA SECONDA ECCEZIONE*, IL DEBUGGER HA UNA 2ª POSSIBILITÀ DI GESTIRLA

In effetti, postrio un'eccezione dentro il gestore delle eccezioni è problematico.

⇝ Le eccezioni sono molto usate dai malware per verificare se c'è un debugger in esecuzione, o per mascondere il flusso di esecuzione.

* OPPURE IL PROGRAMMA NON È IN GRADO DI GESTIRE L'ECCEZIONE.
IN TAL CASO SI TRATTEREBBE DI UN BUG DELL'APPLICAZIONE (DEL MALWARE) CHE NE CAUSA IL CRASH

Tracce:

Olly DBG è in grado anche di memorizzare le tracce di esecuzione (≡ tutti gli stati che il debugger ha incontrato eseguendo le istruzioni macchina).
↳ GLI STATI AD ESEMPIO COMPRENDONO I VALORI ASSUNTI DAI VARI REGISTRI CON LA LORO EVOLUZIONE.
→ Una volta registrata la traccia, può essere rieseguita.

→ MOVS ≡ memcpy [REP MOVSB]
→ CMPS ≡ memcmp [REP CMPSB]
→ SCAS ≡ RICERCA DI UN BYTE ALL'INTERNO DELLA STRINGA [REPNE SCASB]
→ STOS ≡ memset [REP STOSB]

→ la B finale sta per matrice, vale una; byte (in altre = double)

11/11/2021

→ Le istruzioni di stringa (come REPNE SCAS) sono quasi sicuramente funzioni di libreria che, per motivi di efficienza, sono state scritte in assembly.

18/11/2021

In che modo il malware può scoraggiare l'analista a sfruttare determinati strumenti come il debugger?

1) Identificare se c'è un debugger in esecuzione. (DETECTION)
2) Identificare il comportamento del debugger. (IDENTIFYING DEBUGGER BEHAVIOR)
3) Interferire col comportamento del debugger. (INTERFERING WITH DEBUGGER)
4) Sfruttare le vulnerabilità del debugger. (EXPLOITING DEBUGGER VULNERABILITIES)

L'approccio ① può essere realizzato:

→ Con delle API (e.g. IsDebuggerPresent, CheckRemoteDebuggerPresent).
↳ Possono anche non comparire nella IAT (Import Address Table = tabella delle funzioni invocate nel programma; viene istanziata in fase di compilazione).
↳ In tal caso, possono essere sfruttate le API LoadLibrary e GetProcAddress; LoadLibrary carica a run-time nello spazio di indirizzam. del processo una data DLL, le cui funzioni in essa contenute appaiono così la IAT.

NtQueryInformationProcess, OutputDebugString
↳ ANCHE

CheckRemoteDebuggerPresent può individuare la presenza di un debugger anche sugli altri processi che girano sulla stessa macchina.

→ Con l'accesso diretto alla struttura PEB (Process Environment Block), che si trova all'offset 0x30 del segmento puntato dal registro FS.
All'offset 2 di PEB c'è un campo chiamato BeingDebugged.
↳ O MEGLIO AL BIT DI OFFSET 7

All'offset 0x18 di PEB c'è un campo chiamato ProcessHeap, che descrive come è fatto l'heap iniziale del processo. ∃ due flag (ForceFlags, Flags) che sono diversi da zero se c'è un debugger; tuttavia, il loro offset all'interno del ProcessHeap varia al variare della versione di Windows.

Questo campo indica come viene creato l'heap iniziale del processo.

∃ un altro campo all'offset 0x68 di PEB (NtGlobalFlags) che può accorgersi se c'è un debugger in esecuzione (e il debugger c'è nel momento in cui questo campo vale 0x70).

→ Con la ricerca dei residui nel sistema (ovvero di determinati programmi usati da gli analisti all'interno del file system): se è presente OllyDbg o simili, il malware può comportarsi diversamente). → Attraverso al file system il malware può anche decidere di effettuare una scansione alla registry key.

→ Con la ricerca di un processo debugger in esecuzione contemporaneamente al malware, a prescindere su chi sta lavorando il debugger (→ API FindWindow).
↳ FindWindow("OllyDbg", 0) verifica se ∃ aperta una finestra chiamata "OllyDbg".

L'approccio ② può essere realizzato andando a vedere i breakpoint utilizzati, ovvero:

→ Con INT3 scanning (≡ ricerca dell'istruzione che solleva un debug exception).
↳ Questo approccio presenta 2 problemi: FALSI POSITIVI (byte 0xcc non relativo a INT3) e FALSI NEGATIVI (uso di INT 3 anziché INT3)

→ Con il checksumming, che controlla se il codice in memoria viene modificato (il che, di norma, viene fatto solo dal debugger).

→ Con il timing check, che controlla in quanto tempo viene eseguito il programma (istruz. RDTSC), API QueryPerformanceCounter, GetTickCount). Oppure in quanto tempo viene nuovo gestite determinate eccezioni.
↳ IN TUTTI I 3 I CASI, L'ISTRUZ./L'API VIENE INVOCATA 2 VOLTE E VIENE FATTO UN CONFRONTO TRA TIMESTAMP RESTITUITI.

L'approccio ③ può essere realizzato:

→ Controllando se c'è un debugger in esecuzione anche prima della 1° istruzione dell'entry point: alcune volte, il debugger può eseguire delle istruzioni TLS (TLS callback), ancora prima di passare per l'entry point. OVE serve a popolare le sezioni TLS dell'eseguibile

→ Con il SEH (Structured Exception Handling), ovvero ad esempio guardando se viene eseguito il gestore dell'eccezione relativo all'istruz. INT3 (non viene eseguito in modalità debug).

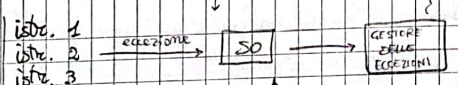
→ Inserendo degli interrupt / dei breakpoint (come l'istruz. INT 3) con lo scopo di rallentare o confondere il debugger, oppure proprio con lo scopo di vedere se c'è un debugger che gestisce queste eccezioni oppure no.

23/11/2021

Ma come funzionano le eccezioni in Windows?

L'eccezione viene dirottata verso il SO

Può essere:
→ Definito dal programmatore
→ Fornito dal SO (gestore di default) → SPESSO TERMINA IL PROCESSO



Il SO non sempre è in grado di gestire l'eccezione; quando non lo è, dirotta l'eccezione verso il gestore delle eccezioni, che è scritto nel programma.

Non esiste un solo modo per gestire le eccezioni in Windows: ne abbiamo almeno 5.

- SEH 1 → Molto antico e molto raro
- SEH 2 → Ancora + raro di SEH 1
- SEH 3
- SEH 4

  → Spesso indicati con SEH; sono gestori definiti dalla Microsoft per gestire le eccez. in C
- SEH 5 → Spesso indicato con EH; è un gestore definito dalla Microsoft per gestire le eccez. in C++

Parliamo un po' di SEH 3... (SEH = Structured Exception Handler)

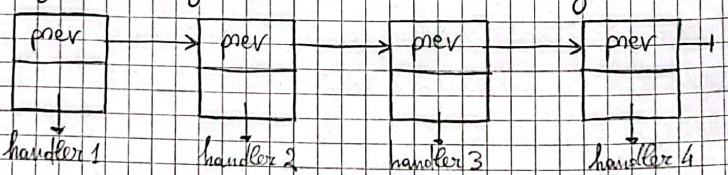
```
• __try {
      blocco di codice
  }
  __except (condition) {
      altro blocco di codice
  }
```

```
__try {
    blocco di codice
}
__finally {
    altro blocco di codice
}
```

↳ Questo costrutto può anche essere ricorsivam. innestato (nidificato).

Ha una struttura a stack nel senso che l'ultimo gestore inserito sarà il primo interpellato x la gestione delle eccez.

In SEH3 i gestori sono organizzati tramite una lista collegata:



Lui è il primo gestore che viene invocato qual viene sollevata un'eccezione; se è in grado di gestirla, ok; altrimenti viene "interrogato" l'handler successivo; e così via.

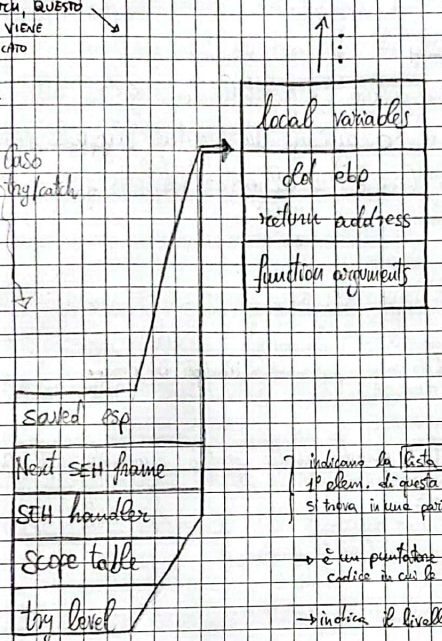
Se neanche l'ultimo handler può gestire l'eccezione, ~~...~~ questo ultimo sarà anche l'ultimo handler
⇒ Dovrebbe essere qualcuno gestito dal gestore di default.

Consideriamo ora lo stack quando viene invocata una funzione:

QUANDO C'È UN BLOCCO TRY/CATCH, QUESTO BLOCCO VIENE CONTRASSEGNATO

5.

caso try/catch




```
struct EXCEPTION_REGISTRATION {
    DWORD prev;
    DWORD handler; }
```

indicano la lista collegata di cui parlavamo; non è detto che siamo il 1° elem. di questa lista. L'info su dov'è il 1° elem. della lista (-> il 1° gestore) si trova in una particolare struttura. *

→ è un puntatore a una tabella contenente record indicanti i blocchi di codice in cui le sezioni try agiscono.

→ indica il livello di nidificaz. del blocco try.

* TEB (Thread Environment Block)

TIB (Thread Information Block)

OFFSET 0: puntatore alla SEH chain (≡ lista collegata)

← IL REGISTRO FS PUNTA PROPRIO A QUESTO

Per aggiungere un'eccezione, bisogna dunque:

(1) → Aggiungere un nuovo elemento E della lista collegata dei gestori (dopo averne definito l'handler).

(2) → E.prev = FS:[0]

(3) → FS:[0] = E

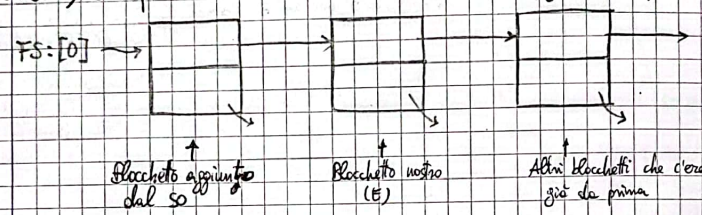
OVVERO:

(1) → push Exception Handler

(2) → push FS:[0]

(3) → mov FS:[0], esp → che, di fatto, è l'indirizzo di E essendo quest'ultimo l'ultimo elemento aggiunto sullo stack.

Normalmente, quando definiamo un nuovo gestore, è possibile dover gestire altre eccezioni all'interno del blocco try/catch. Perciò il SO aggiunge un nuovo blocchetto (un nuovo elemento) che punta al nostro nuovo elem. E e al gestore definito dal SO.



→ Per rimuovere il nostro blocchetto E (e quindi anche quello aggiunto dal SO), sono necessarie le seguenti istruzioni:

```
MOV ESP, [ESP+8]
MOV EAX, FS:[0]
MOV EAX, [EAX]
MOV EAX, [EAX]
MOV FS:[0], EAX
ADD ESP, 8
```

Due parole su SEH4...

↳ Qui, sullo stack, tra SAVED ESP e LOCAL VARIABLES viene aggiunto un valore, detto GS cookie. Esso serve per identificare eventuali buffer overflow (vedi prima me lezioni di Quaglia).

IL VALORE GS COOKIE AGGIUNTO SULLO STACK PUÒ ESSERE UTILE PROPRIO A QUESTO

Il malware può interferire col funzionam. dei debugger sfruttando tutto questo (in particolare aggiungendo dinamicam. nuovi gestori alla SEH per cambiare il suo stesso flusso di esecuzione).

(*) Indica il numero di entry dell'array DataDirectory (che si trova subito dopo all'interno dell'header); tale array (composto al più da 16 entry) indica dove si trovano gli altri componenti dell'eseguibile all'interno del file.
→ QUALUNQUE VALORE MAGGIORE DI 16 È INVALIDO E FA CRASHARE LE PRIME VERSIONI DI OLLYDBG.

25/11/2021

Analizziamo ora l'approccio (4) che può essere adottato dal malware per ostacolare i debugger e, in particolare, OllyDbg.

Qui viene impiegato il comp. NumberOfRvaAndSizes (*) ora questo è fatto sistem.

→ Sfruttando i bug del PE header, che possono essere:

- IMAGE_OPTIONAL_HEADER → Cambiando qualche valore nella sez. optional header della tabella (rendendo sbagliato il checksum), l'esecuz. normale del malware va a buon fine mentre le prime vers. di OllyDbg crashavano
- IMAGE_SECTION_HEADER → Stesso discorso di IMAGE_OPTIONAL_HEADER (dovuta nello specifico viene interrotto size of raw data)
- API OutputDebugString() → Serve a inviare a un debugger un messaggio in formato printf-like. Usa stringa di tipo "%s%s..%s%s%s" in modo che il debugger dello accedere sullo stack a un indirizzo non valido (con un num. sufficiente di %s) → OllyDbg va in segfault
  → È stato sistemato solo nelle versioni 2.x di OllyDbg

Ma in che modo il malware può scoraggiare l'analista a sfruttare i disassemblatori? Per rispondere a questa domanda, richiamiamo prima i tipi di disassemblatori:

→ LINEAR DISASSEMBLER (e.g. objdump, gdb): iniziamo l'analisi a partire dall'entry point e procediamo "linearmente".
Qui è bene distinguere gli instruction set (ISA) in:

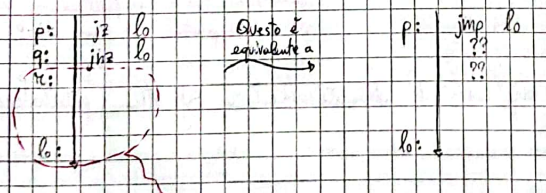
Qui i disassemblatori lineari si comportano meglio

- RISC-like ISA → tutte le istruzioni macchina sono composte dallo stesso numero di byte.
- CISC-like ISA → le istruz. macchina hanno lunghezza variabile (che si capisce dal 1° byte delle istruz. stesse).

→ FLOW-ORIENTED DISASSEMBLER (E disassemblatori interattivi).
- PUNTI CRITICI: - In caso di salto condizionato viene disassemblata prima la porzione di codice relativa alla condiz. vera o a quella falsa? [CONTROLL. STATICO]
- In caso di CALL viene disassemblata prima la funzione o viene finito prima il chiamante? [CONTROLL. IL CHIAMANTE]

L'obiettivo del malware contro i disassemblatori è fare in modo che questi ultimi sbaglino ad analizzare l'eseguibile e generino un codice "errato".

↳ Vediamo come:



Ma com'è possibile? Basta che l'autore del malware aggiunga un byte spazzo (= non appartenente ad alcuna istruzione macchina, come ad esempio una short o un char) subito dopo la seconda istruz. di salto condiz. In tal modo, Ghidra lo interpreta come un byte** appartenente a qualche istruzione, rischiando di disallineare tutto ciò che viene dopo. I byte + usati a questo scopo sono E9 (inizio di una jmp) ed E8 (inizio di una call).

INDIRIZZI

Qui Ghidra sbaglia ad analizzare a causa di un disallineam. delle istruzioni; il punto è anche che assume che ci siano x forza delle istruz. macchina dopo un salto condizionato (cosa che chiaramente non vale per un salto non condizionato).

** ROGUE BYTE

Variante del caso: xor eax, eax — jz lo ⟶ (Questo è equival. a) jmp lo

??
??

L'errore deriva dal fatto che, in questo caso, non esiste alcun codice relativo alla condizione falsa del salto (creandosi da un salto un'istruz. di lunghezza sei mezzi sopra).

Di nuovo, qui il disassemblatore sbaglia.

Il punto è che, come dicevamo, il disassemblatore scelse di disassemblare per primo il codice relativo alla condizione falsa del salto e, in caso di conflitto, si fida più di quello.

C'è una tecnica ancora + insidiosa: il DISASSEMBLAGGIO IMPOSSIBILE.

ESEMPIO:

```
jmp -1

EB   FF   C0   48   ...
     EFFETTO DELLA JUMP

     inc eax   dec eax
```

Il codice prosegue qui come se EB FF C0 48 non ci fosse; tuttavia, tale sequenza, pur non facendo niente a livello di registri e di istruz., confonde il disassemblatore.

Effettivam., il disassemblatore non assume mai che un qualche byte (come FF nel nostro caso) possa far parte di due istruz. differenti; il processore sì (ed è per questo che il programma continua a funzionare correttamente).

ALTRO ESEMPIO:

```
EFFETTO DELLA JUMP

mov AX, 0x05eb   xor EAX,EAX   jz jmp -6

66   B8   EB   05   31   C0   74   FA   E8   ...

EFFETTO DELLA JUMP

jmp +5
```

Qui il disassemblatore sbaglia: E8 si trova dopo un salto condiz. che, in realtà, condizionato non è; in particolare, E8 viene interpretato con una call.

Un altro modo per mettere in difficoltà il disassemblatore sfrutta i puntatori a funzione (giocandoci).

e.g. mov [ebp+var_4], indirizzo_funzione
...
call [ebp+var_4]

Se tra mov e call vengono fatte delle operazioni che si mantengono tra loro su indirizzo funzione, Ghidra non ci capisce comp. + niente e vien fuori di altro che a indirizzo funzione mette una funzione.

30/11/2021

Vediamo altre tecniche di disassemblaggio attuate dal malware.

→ ABUSO DEL VALORE DI RITORNO

```
        CALL  l0
l0:     ADD [ESP], 5
        RET
l0+5:
```

Questo insieme di istruzioni è equivalente a una NOP

Ma dal punto di vista del disassemblatore è cambiato tutto: in particolare, se all'indirizzo l0+5 inizio una funzione vera e propria, non viene riconosciuta come tale da Ghidra perché viene sto accedendo tramite l'istruzione di RET; inoltre, in tal modo vengono anche persi i riferimenti ai parametri di input.

→ GESTIONE DELLE ECCEZIONI STRUTTURATE: si può alterare il flusso di esecuzione visto dal disassemblatore (ad esempio aggiungendo un'istruzione di return che in realtà non viene mai invocata, ma fa pensare al disassemblatore che la funzione termini con quella return).

↳ ESEMPIO IN CUI QUESTA TECNICA FUNZIONA:

```
                PUSH  EAX ⇒ handler_address
                PUSH  FS:[0x0]
                MOV   FS:[0x0], ESP
                XOR   EAX, EAX
                DIV   EAX                  ← Divisione per 0: eccezione!
                RET
                LEA   ESI, [ESI]           ← Dopo la gestione dell'eccezione, in realtà, si riprende il flusso
handler_address: MOV  ESP, [ESP+0x8]       ← da qua, ma Ghidra assume che si riprenda dalla return.
                MOV   EAX, FS:[0x0]
                MOV   EAX, [EAX]
                MOV   EAX, [EAX]
                MOV   FS:[0x0], EAX
                ADD   ESP, 0x8
```

Dunque, da qui in poi, Ghidra in realtà sbaglia a interpretare l'appartenenza delle istruzioni alla funzione in cui ci troviamo: in particolare, pensa che l'appartenenza non ci sia per via di quella istruzione di RET.

⇒ Tutto sto cinema è "quasi" equivalente a una NOP

→ ~~UTILIZZO DI if DETERMINISTICI~~ IMPEDIMENTO DELL'ANALISI DELLO STACK FRAME TRAMITE L'UTILIZZO DI if DETERMINISTICI

```
        SUB  ESP, 8
        SUB  ESP, 4
        CMP  ESP, 0x1000
        JL   l0
        ADD  ESP, 4
        JMP  l1
l0:     ADD  ESP, 0x104
l1:     <accessing stack>
```

→ Sto aggiungendo a ESP un valore grande ⇒ sto andando a prendere valori sullo stack relativi ad altre funzioni

* OVVERO: è praticamente impossibile che lo stack si trovi nella 1ª pagina dell'address space del processo (questo in tutti i casi).

Per l'applicazione in sé questo non è un problema perché questo ramo di esecuzione mai viene preso mai (è impossibile che ESP < 0x1000)*, per cui mai viene mai eseguita l'ADD ESP, 0x104. Tuttavia, Ghidra non è in grado di fare tale ragionamento e tiene aperte entrambe le opzioni; quindi cercherà di prendere valori sullo stack al di fuori dello stack frame, Ghidra deve assumere che ci siano MOLTI + parametri di questa operazione presente.

ULTIMAMENTE VENGONO PRESE SEMPRE PIÙ
→ PERCHÉ ANCHE GLI UTENTI "NORMALI" STANNO INIZIANDO A FAR USO DELLE VM.

Contromisure contro le macchine virtuali:

→ VmWare
QEMU
Hyper-V
Virtual Box

Il malware può anche chiedersi se sta girando su una macchina fisica o su una VM, e comportarsi di conseguenza.

ad esempio non eseguire, cercare la vulnerabilità della VM, andare a infettare la macchina fisica.

Vediamo quali sono le tecniche anti-VM.

→ ESECUZIONE DELL'ISTRUZIONE cpuid CON LO SCOPO DI CAPIRE QUAL È IL VENDITORE (E LA MARCA) DEL PROCESSORE, che viene rappresentata tramite una stringa, che potrebbe essere differente tra macchina fisica e macchina virtuale (non in Virtual Box).
Se passiamo il valore 0 a cpuid
Un altro scopo dell'invocazione a cpuid (stavolta con parametro 1) è ottenere un centinaio di flag sulle info della CPU; uno di questi (il 31°) è inutilizzato sulla macchina fisica, ed è pari a 1 solo quando si utilizza una macchina virtuale.
Un terzo scopo (utilizzando il parametro 0x40000000) è ottenere il vendor della VM.

→ TECNICHE BASATE SUI MAC ADDRESS: Sappiamo che ciascun costruttore di schede di rete ha un suo prefisso di indirizzi MAC per le schede di rete che produce. Alcuni prefissi sono relativi specificatamente alle VM (e.g. 08-00+27-.. è di Virtual Box).

→ INTERROGAZIONE DELLE REGISTRY KEY: le VM possono lasciare delle tracce del fatto che l'hw è emulato all'interno del registro di sistema.

→ RICONOSCIMENTO DELL'ESISTENZA DELLE ESTENSIONI (O DEI SERVIZI) PER LE MACCHINE VIRTUALI.
↳ È possibile cercarle nel registro di sistema, tra i servizi attivi oppure tra i file all'interno del sistema.

→ TECNICHE BASATE SULLE ISTRUZIONI VULNERABILI: ∃ delle istruzioni che servono al SO (come quelle che vengono eseguite in Kernel mode) ma possono anche essere tranquillam. eseguite in user mode e, per questo motivo, è necessario che siano emulate.
Alcuni esempi di queste istruz. sono: CPUID, SIDT, SGDT, SLDT
Interrupt Descriptor Table
Global Descriptor Table
Local Descriptor Table

È con queste istruz. che il malware cerca di capire se sta girando in una VM, perché quest'ultima non può permettersi di lasciare tali istruz. identiche alla macc

→ SFRUTTAMENTO DEGLI INDIRIZZI DI INPUT/OUTPUT (IN, OUT)

```
MOV EAX, 'VMXh'
MOV EBX, addr
MOV ECX, 0Ah
MOV DX, 'VX'
IN EAX, DX
```

Il malware, con queste istruzioni, sta cercando di ottenere una risposta da una macchina virtuale

↳ Nel caso in cui il malware girasse su una macchina fisica, l'istruz. IN non ha effetto.

Sandbox:

È un approccio x l'analisi del malware che consiste semplicemente nel far eseguire il malware all'interno di una VM e nel registrare tutto ciò che fa.

↳ Chiaram. può essere vittima delle tecniche anti-VM.

02/12/2021

Packer:

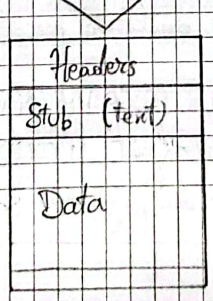
Consideriamo un eseguibile originale:

È costituito da due tabelle in memoria:
- La 1ª è una lista di nomi/ordinali usati dal loader o dalla stub per determinare quali funzioni sono necessarie.
- La 2ª è una lista degli indirizzi di tutte le funzioni importate.

Mentre il programma è in esecuzione, solo la 2ª tabella è necessaria, per cui il packer può rimuovere la 1ª e lasciare così la IAT intatta.

INDICATORI DI UN PROGRAMMA IMPACCHETTATO:
- → Presenza di pochissimi import.
- → Pochissimo codice disassemblato su Ghidra.
- → Warning di OllyDbg sulla possibilità che il programma sia stato impacchettato.
- → Sezioni del programma insolite.
- → Dimensioni delle sezioni anormali (e.g. text con SIZE OF RAW DATA = 0 ∧ VIRTUAL SIZE > 0).
- → PE:ID.
- → Entropia delle sezioni text, data più alta del normale.

File modificato → in cui non c'è nulla in chiaro del file originale



→ Codice che serve a ritrovare le istruzioni e i dati originali dell'exe

→ Include tutte le informazioni (non in chiaro) del file originale

macchina fisica.

Lo stub:

1) Spacchetta la sezione "data".
2) Risolve gli import (ovvero i collegam. alle DLL originali).
3) Salta all' OEP (Original Entry Point). → IL SALTO È SETTO tail jump

Il problema grosso è che ∃ tanti tipi di impacchettatori differenti che compattano i programmi in modo diverso, risolvono gli import in modo diverso e saltano all'OEP in modo diverso. Tra l'altro, i packer sono estendibili: + eseguibili impacchettati dallo stesso packer possono essere trattati in modo diverso.
Non esistono dunque spacchettatori universali.

STRATEGIE PER RISOLVERE I 3 PASSI ESEGUITI DALLO STUB:

1) Su questo non ci sono particolari problemi (allocazione in memoria di spazio destinato alle sez. del file originale + copia dei dati in tali aree di memoria x ottenere l'eseguibile spacchettato)
2) Per quanto riguarda la risoluzione degli import, si hanno + possibilità:
   → Mantenere la IAT originale nel file impacchettato; questo lascia informazioni all'analista e dal punto di vista del malware non va bene.
   → Mantenere la IAT originale ma collegata a una sola API per ogni DLL.
   → Non tenere (quasi) nulla della IAT originale: lo stub deve così ricalcolare a mano tutti gli indirizzi nella DLL originale. Questo si fa con le API LoadLibrary e GetProcAddress.
   → Non tenere nulla della IAT originale e facendo in modo che lo stub invochi LoadLibrary e GetProcAddress senza sfruttare la IAT (che stavolta non mantiene i nomi, fornendoli necessari a LoadLibrary e GetProcAddress).
3) Il salto all'OEP può essere fatto banalmente con una jump, eventualm. applicando le tecniche anti-disassembler analizzate precedentemente, o anche con le funzioni NtContinue, ZwContinue.

IDEA SU COME EFFETTUARE L'ANALISI DI UN PROGRAMMA IMPACCHETTATO:
Utilizzare un debugger, inserire un breakpoint in corrispondenza dell'istruz. di salto verso l'OEP e salvarsi il dump del programma spacchettato che è stato ap=

TECNICHE CHE UN ANALISTA PUÒ SFRUTTARE PER L'UNPACKING
- → Static automated unpacking
- → Dynamic automated unpacking
- → Dynamic manual unpacking (NOI STIAMO APPRENDENDO QUESTA TECNICA)

pena caricato in memoria.

Questo approccio ha però due problemi:

1) Trovare l'istruz. di salto verso l'OEP non è banale.
2) Bisogna ancora ricostruirsi a mano la IAT.

PERMETTE DI SKIPPARE ALMENO LA PARTE IN ZIALE DELLO STUB, CHE È LA + COMPLICATA.

1) ∃ delle strategie per trovare l'OEP:
- Salto tra sezioni differenti.
- Salto molto lontano. → SICURAM. NON FUNZIONA NEL CASO IN CUI LO STUB, TRA LE ALTRE COSE, SI AUTOMODIFICA
- Utilizzo di tool (e.g. OllyDump in OllyDbg). → ANCHE IL COMANDO RUN TRACE DI OLLYDBG PUÒ ESSERE UTILE.
- Inserimento di un breakpoint alla fine di ogni ciclo (sperando di beccare il punto in cui si salta verso l'OEP). → ALL'INTERNO DEI CICLI.
  CIÒ PERMETTE DI ESEGUIRE PASSO PASSO L'INTERO STUB SENZA PERÒ RIMANERE BLOCCATI
- Inserimento dei breakpoint nelle chiamate alle API importate e sicuramente utilizzate dal SO prima di eseguire il main.
- Inserimento dei breakpoint sullo stack (≡ sulle istruzioni di accesso allo stack).
  ↳ SONO BREAKPOINT DI ACCESSO ⇒ SONO BREAKPOINT HARDWARE OPPURE MEMORY BREAKPOINT
  ↳ IN PARTICOLARE IL BREAKPOINT VA POSTO AL PRIMISSIMO ELEMENTO ALCUNO SULLO STACK PERCHÉ AL CI SI ASPETTA CHE VENGA EFFETTUATA LA PUSH DI TALE ELEMENTO SUL PILA FINE DELLO STUB.

07/12/2021

Dopo aver ~~identificato~~ salvato il dump del programma spacchettato, come accennavamo, è necessario ricostruire la Import Address Table.

Riassumiamo schematicamente i passaggi che deve seguire l'analista:

1) Identificare l'OEP.
2) Eseguire lo stub fino all'OEP.
3) Salvare il memory dump.
4) Ricostruire la IAT (può essere un passaggio difficile o addirittura impossibile).

Per ricostruirla A MANO bisognerebbe (dopo aver ottenuto l'exe unpacked):
- Cercare su Ghidra tutte le CALL ex FC.. Jxxxxfd.
- Cercare tali indirizzi su OllyDbg x vedere a quale funz. corrisponde.
- Istruire Ghidra riguardo l'entità di tali funzioni.

↳ CIÒ COMPORTERÀ FARE AN. STATICA SULL'UNPACKED E AN. DINAMICA SUL PACKED**

Comunque anche senza di questo si può fare un po' di analisi statica*

→ Un tool molto usato per ricostruire la IAT è ImpREC (≡ Import REConstructor).

Windows Defender lo odia a morte perché dal suo pto di vista, modifica i programmi che si trovano sull'hard disk.

* In effetti in tal caso otteniamo un programma in chiaro e non funzionante

** PURTROPPO NON È QUESTO IL MODO CHE PERMETTE DI FIXARE LA IAT SULL'ESEGUIBILE UNPACKED: FARE CIÒ PUÒ ESSERE MOLTO + TEDIOSO.

Quello che si fa con ImpREC è:

1) Lanciare il programma impacchettato (senza OllyDbg però); il processo, però, deve rimanere attivo (per cui spesso si hanno problemi).
2) Selezionare il processo in ImpREC.
3) Inserire l'OEP in ImpREC.
4) Cliccare su IAT AutoSearch.
5) Cliccare su Get Imports.
6) Controllare la validità di ciò che ImpREC ha trovato.
7) Cliccare su Fix Dump.

PACKER PIÙ COMUNI:
- UPX
- PE COMPACT
- ASPACK
- PETITE
- MINIPACK
- THEMIDA

Le macchine infette presen[illegible] ancora vuote [illegible] bombardare [illegible] stanno conver[illegible] (dovuy a servi[illegible]

09/12/2021

Classificazione del malware:

① Classificazione basata su che cosa fa il malware.
② Classificazione basata su qual è la base d'attacco del malware.
③ Classificazione basata sui meccanismi interni del malware.

① (I) ADWARE: ha lo scopo di mostrare le pubblicità sul PC dell'utente senza che l'utente stesso lo abbia richiesto; in alternativa può sostituire le ricerche e/o i risultati delle ricerche che l'utente effettua su un motore di ricerca; può anche collezionare informazioni sugli interessi commerciali dell'utente.

(II) MALVERTISING: quando l'utente clicca su una finestra di pop-up relativa a una pubblicità, fa in modo che venga aperto un sito relativo a una pubblicità differente. ②

(III) SPYWARE: ha lo scopo di spiare l'utente ed estrapolarne le informazioni sensibili. Viene detto anche INFORMATION STEALER. Un esempio di programma con tale finalità è il Keylogger; un secondo tipo di programma è l'hasher, che non fa altro che rilevare l'hash delle informazioni sensibili

dell'utente piuttosto che le informazioni sensibili direttamente in chiaro.
Spyware più utilizzati nel 2019: LOKIBOT – FORMBOOK – AZORULT – PREDATOR THE THIEF – VIDAR.

(IV) BACKDOOR, che può essere: → DÀ LA POSSIBILITÀ AL MALINTENZIONATO DI ACCEDERE DA REMOTO SUL SISTEMA INFORMATICO DELLA VITTIMA.

- Backdoor semplice, che può essere usata col comando netcat (nc), oppure può essere creata con un programmino Windows.
- Bot / Botnet, che è una rete di macchine infettate (che tipicam. sono server) a cui possono essere inviati comandi da remoto in modo da controllare queste macchine.

  Le macchine infette possono essere usate per bombardare uno stesso server (denial of service)
- RAT (Remote Access Trojan), che è simile al Bot, con la differenza che non si vuole infettare + server possibili, bensì solo determinati server ben precisi. RAT più utilizzati nel 2019: NANOCORE – NJRAT – BLACKBINN – REMCOS – ADWIND.

(V) RANSOMWARE: ha lo scopo di cifrare i file dell'utente, in modo tale che, per essere decrittati, è necessario pagare una somma di denaro a chi ha scritto il malware (o chi per esso). Ransomware più utilizzati nel 2019: SHADE – GAND CRAB – WANNA CRY – SODINO KIBI – DHARMA.

(VI) CPU CYCLES STEALER: ha lo scopo di caricare maggiormente la CPU del PC dell'utente, facendogli spreccare più energia.

(VII) LOADER: ha lo scopo di installare un altro programma che è a sua volta malware; è interessante perché si basta di un eseguibile molto piccolo.

② (I) VIRUS: è un ~~programma~~ "modulo" in grado di riprodurre se stesso; tipicamente è una porzione di codice che entra a far parte di un programma legittimo ed entra in azione quando il programma legittimo viene avviato. I virus possono infettare i file eseguibili, i settori di boot dell'hard disk, il BIOS (cioè il firmware), ecc.

(II) WORMS: è un programma standalone che può riprodurre se stesso; a differenza del virus, è "autosufficiente" (≡ è un programma completo).

(III) TROJAN: è un programma malevolo che viene spacciato come utile o legittimo; spesso (ma non sempre) è un loader. Trojan più utilizzati nel 2019:
EMOTET – AGENTESLA – HAWKEYE – AHADEY – BETBOT.

(IV) PHISING: consiste in tutte quelle pratiche di ingegneria sociale per cui l'utente vittima è indotto a fare qualcosa che causa danni a se stesso.
Una sottocategoria del phising è lo SPEAR PHISING, che consiste nelle pratiche di ingegneria sociale che hanno una vittima ben precisa nel bersaglio.
Un'altra sottocategoria è lo SCAREWARE, che fa in modo di generare ansia all'utente, convincendolo a scaricare un qualche software (che, appunto, può essere malware).

③ (I) ROOTKIT: è codice installato a livello di sistema operativo, utilizzando i meccanismi di base del sistema operativo per mascondersi e per rimanere persistente. Un rootkit può essere KERNEL-SPACE oppure USER-SPACE.

(II) APP-BASED: è codice installato a livello di una qualche applicazione.

ROOTKIT CHE OPERA DI PRIVILEGI DI LIVELLO UTENTE; AD ESEMPIO PUÒ INIETTARE UNA LIBRERIA A UN'APPLICAZIONE IN ESECUZIONE, OPPURE PUÒ SEMPLICEMENTE SOVRASCRIVERE LA MEMORIA DELL'APPLICAZIONE.

14/12/2021

Meccanismi utilizzati dal malware (≡ obiettivi tattici):

→ PERSISTENZA
→ PRIVILEGI
→ SEGRETEZZA
→ EVASIONE

Persistenza:

La si può ottenere coi seguenti meccanismi:

- Utilizzando una particolare registry key (HKLM/SOFTWARE/Microsoft/Windows/CurrentVersion/Run) che si occupa degli eseguibili da lanciare all'avvio della macchina.

TUTTE LE DLL X CHE GIACCIONO DI QUESTO TIPO DI REGISTRY FANNO LETTE APPINIT_DLL

- Con AppInit_DLL, che esiste nel caricare automaticamente una DLL X in ogni applicazione che usa User32.DLL; in questo caso, il malware vive all'interno della DLL X e viene lanciato ogni volta che viene eseguito un programma che contiene User32.DLL.
  Notiamo che le DLL non sono troppo diverse dalle normali applicazioni, ma hanno la particolarità di avere tanti punti di ingresso. → UNO PER OGNI API SOTTOLINEA VIELLA DLL
  ↳ Il malware viene definito nella DLL Main che rappresenta un export della DLL. → IL CODICE DI DLL MAIN È QUELLO ESEGUITO APPENA LA DLL VIENE COLLEGATA A UN PROCESSO

  Esiste un'altra registry Key apposita che tiene traccia delle DLL che vengono caricate automaticamente. → HKLM/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Windows
  ↳ (Q AppInit_DLL)

- Con WinLogon Notify: Windows permette di caricare automaticam. una DLL ogni volta che si scatena uno dei seguenti eventi: log on di un utente, log off di un utente, shutdown, startup e lock screen.
  ↳ TALE DLL VIENE SCRITTA NEL REGISTRO HKLM/SOFTWARE/MICROSOFT/Windows NT/CurrentVersion/WinLogon/Notify

- Utilizzando i servizi di Windows (o i demoni di Unix): il malware può tentare di installarsi come servizio all'interno della macchina della vittima, e può farlo installandosi nelle registry Key CurrentControlSet, ControlSet001 e simili.
  ↳ Queste registry Key contengono le configurazioni del SO che, tra le altre cose, indicano quali servizi vengono attivati automaticam. all'avvio della macchina.

  ∃ dei servizi DLL-based: SVCHOST.EXE permette di eseguire delle DLL che contengono un qualche servizio (ricordiamo che le DLL non possono essere lanciate così come sono, bensì devono "attaccarsi" a un qualche processo).
  → I PROCESSI SVCHOST.EXE POSSONO ESSERE PIÙ DI UNO E POSSONO ESSERE RELATIVI A MODULI DLL; SI PARLA COSÌ DI GRUPPI DI SERVIZI.

- Sfruttando i Trojanized System Executable (o Binary), cioè modificando degli eseguibili all'interno del sistema affinché eseguano malware; tuttavia, questo meccanismo è facilmente sgamabile grazie a un tool che conosce le firme digitali degli eseguibili del sistema.
  ↳ Comunque, un esempio di qst meccanismo è il DETOUR, che consiste in un'i=

istruzione di jump posta vicino all'entry point di un eseguibile che porta a eseguire codice dal l'altra parte, che magari può consistere in malware (che ovviamente non è codice ufficiale del programma; magari originariamente c'era packing).

Vediamo questo in pratica:

```
JMP L0
L2:
PUSH EBP
MOV EBP, ESP
XOR EAX, EAX
.db "malware.dll", 0
```

sostituisce

A distanza 24h dall'istruzione di JMP

```
L0: PUSHA
CALL L1
NOP
POP ECX
MOV EAX, ECX
ADD EAX, 24h
PUSH EAX
ADD EAX, 0xxfhh
MOV EAX, [ECX]
ADD EAX, 0xxyyh
CALL EAX
POPA
PUSH EBP
MOV EBP, ESP
XOR EAX, EAX
JMP L2
```

PUSHA → SALVARE TUTTI I REGISTRI SULLO STACK PRIMA DI RI PRENDERE POI LA NORMALE ESECUZIONE DELL'APPLICAZIONE SENZA CORRUZIONE

CALL EAX → LoadLibraryA

Da qui si ricomincia con la normale esecuzione del programma patchato

Questo meccanismo porta al caricamento di malware dll ogni qual volta che il programma patchato viene eseguito

↳ Ricordiamo che il caricamento

ATTENZIONE: questo meccanismo, con qualche lieve ~~modifica~~ aggiustamento, non porta più alla modifica del file eseguibile, bensì può portare alla modifica del processo in esecuzione.

- Utilizzando dei meccanismi DLL-load order (Hijacking): ∃ degli algoritmi che permettono di effettuare la ricerca di una DLL nel sistema. In pratica, è possibile, nell'ordine, cercare in:

1) Directory dell'applicazione.
2) Directory corrente.
3) GetSystemDirectory. → API che restituisce la directory .../Windows/System32/
4) 16-bit System Directory. → es. .../Windows/System/
5) GetWindowsDirectory. → API che restituisce la directory .../Windows/
6) %PATH%. → scandisce le directory listate nella variabile d'ambiente PATH

Tuttavia, ogni tipo di ricerca è onerosa e "pericolosa". Piuttosto, ci si può basare su delle DLL CONOSCIUTE.

Supponiamo di avere un eseguibile che NON è in %SYSTEMROOT%\System32. (GetSystemDirectory)
↳ PIUTTOSTO È ALTROVE IN %SYSTEMROOT% (cartella /Windows/)

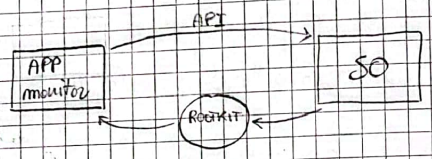
Supponiamo anche che l'eseguibile carichi una DLL NON CONOSCIUTA IN %SYSTEMROOT%\System32 (GetWindowsDirectory).
La vulnerabilità sta nel fatto che, così, il malware può installare (per primo) una DLL omonima in %SYSTEMROOT%\System32 senza alcun problema, e con del codice malevolo all'interno. Così, il SO, nel cercare la DLL, applica l'algoritmo standard descritto precedentemente, per cui cerca prima in Windows (solo poi avrebbe cercato in System32); quando viene trovata (per prima) la DLL malevola in Windows (= %SYSTEMROOT%), viene eseguita quella e la ricerca termina.

Privilegi:
Nel sistema si hanno dei NORMAL USER e l'ADMINISTRATOR. D'altra parte, i processi possono essere USER LEVEL o SYSTEM LEVEL.
→ Per fare in modo che un processo passi a girare con dei diritti di amministratore a partire dai diritti di un utente normale, si devono sfruttare per forza delle vulnerabilità del SO.
→ Per passare da user level a system level, invece, si utilizzano delle particolari API:
- OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &TokenHandle);
  → ACCESS TOKEN = security descriptor per il processo
- LookupPrivilegesValueA(0, "SeDebugPrivilege", &StrutturaDati);
- AdjustTokenPrivileges(&TokenHandle, 0, &NewState, ...);

16/12/2021

Segretezza:

Per conseguire questo obiettivo, il malware utilizza il ROOTKIT, che è un componente che serve a falsare le risposte che il SO dà riguardo la lista dei task, la lista dei servizi, la lista delle connessioni e i file.



Il rootkit può agire in USER MODE o in KERNEL MODE.

Le richieste che arrivano dalle app vengono intercettate dal rootkit e inoltrate da lui al SO. La risposta del SO viene modificata dal rootkit in modo tale da eliminare le informazioni pericolose per il malware

↳ È efficace, perché le app utente non si accorgono di lui

↳ Questo meccanismo richiede che l'applicazione venga modificata affinché catturi il rootkit anziché il SO. Questo lo si fa sfruttando il meccanismo delle DLL: sul disco, quando deve essere invocata un'API definita in una o DLL, il riferimento all'API stesso non è risolto e viene definito direttamente in memoria. Quello che si può fare, è sovvertire il riferimento definito a run-time della funzione che viene invocata, inserendo un indirizzo qualsiasi di una DLL qualsiasi al posto dell'indirizzo dell'API di cui era prevista l'invocazione.

Tale meccanismo è chiamato IAT HOOKING, che può avvenire su file (+ facile sgamabile) oppure in memoria (che a sua volta può avvenire operando sulla IAT oppure tramite un detour; quest'ultima strategia è difficilmente sgamabile).

↳ QUI PARLEREMO DI INLINE HOOKING

Evasione:

I meccanismi per conseguire l'evasione sono detti COVERT LOADING (≡ caricare un qualche eseguibile "mascondendosi"/"fingendo" qualcun altro). Questi meccanismi sono:

- Process injection → DLL injection
  → Direct injection
- Process replacement
- Hook injection
- APC injection

DLL INJECTION:

Tenta di fare eseguire a un processo una DLL che contiene codice malevolo. I passi da seguire sono:

1) Ottenere il PID del processo vittima; lo si fa con le API CreateToolHelp32Snapshot (x OTTENERE LA LISTA DI PROCESSI VITTIMA), Process32First, Process32Next (x ITERARE NELLA LISTA DI PROCESSI).
2) Ottenere l'handle del processo vittima; lo si fa con l'API OpenProcess.
3) Allocare memoria per il processo vittima; lo si fa con l'API VirtualAllocEx.
4) Scrivere nella memoria allocata il percorso (path) della DLL; lo si fa con l'API WriteProcessMemory.
5) Creare un thread nel processo vittima che carica la DLL; lo si fa con l'API CreateRemoteThread(LoadLibraryA, "DLL_name").

DIRECT INJECTION:

Anziché iniettare la DLL in un processo vittima, inietta direttamente il codice (permette maggiorm. al malware di mascondersi ma è fattibile se il codice non è troppo grande).

PROCESS REPLACEMENT:

Il classico target di questa tecnica è svchost.exe che, monitorando i servizi del sistema, lavora a livello di so, per cui può fare quello che vuole. Comunque sia, i passi da seguire sono:

1) Creare un processo nello stato SUSPENDED (→ il processo creato rimane congelato).

2) Invocare l'API ZwUnmapViewOfSection, che sta in NTDLL.DLL (che è l'ultima DLL in vanta prima di entrare in Kernel mode; di fatto è una libreria che contiene delle versioni e proprie system call e permette di fare più cose). Tale API svuota completamente il processo e il relativo address space.

3) Creare tutte le sezioni del processo congelato in modo tale da metterci dentro tutto ciò che si desidera; lo si fa con l'API VirtualAllocEx.

4) Riempire le sezioni del processo congelato con l'API WriteProcessMemory.

5) Ridefinire l'entry point (EIP) del processo congelato con l'API SetThreadContext.

6) Far partire il processo con l'API ResumeThread.

HOOK INJECTION:

Hook = punto del meccanismo di gestione dei messaggi all'interno del sistema dove l'applicazione può installare una subroutine per processare alcuni tipi di messaggi prima che arrivino alla WindowProcedure dell'app stessa.

Consiste nel definire una procedura (ad esempio all'interno di una DLL) che intercetti e gestisca arbitrariamente i messaggi diretti verso qualche processo.

Gli hook si classificano in:

- Hook locali → SETTATI SOLO PER IL PROCESSO SPECIFICO
- Hook remoti → SETTATI LA UN ALTRO PROCESSO
  - di livello alto (DLL) — la subroutine è definita da una DLL che deve dunque essere allegata al processo per il quale l'hook viene attivato.
  - di livello basso (codice della subroutine definito nel processo ~~che mette in piedi un hook~~ per cui lo viene attivato.)

Il meccanismo degli hook è molto usato dai Keylogger, che sfruttano tipicamente gli eventi WH_KEYBOARD, WH_KEYBOARD_LL

→ L'API utilizzata è SetWindowsHookEx (rappresenta un altro modo per iniettare una DLL in un processo).

IN PARTICOLARE LO FA SETTANDO UN HOOK SU UN PROCESSO ESTERNO

APC INJECTION (ASYNCHRONOUS PROCEDURE CALL):

Consiste nel far eseguire a un thread remoto (già esistente) del codice arbitrario (e diverso dall'originale). Ci sono tre casi in cui il SO permette ciò:

→ WaitForSingleObjectEx

→ WaitForMultipleObjectEx

→ SleepEx

Ogni thread, in effetti, ha una coda (LA COSA APC) di procedure che verranno eseguite quando verrà chiamata una delle tre API di cui sopra.

⇝ Abbiamo due varianti di questa tecnica:

- Kernel-mode APC → APC GENERATA PER IL SISTEMA O UN DRIVER
- User-mode APC → API: QueueUserAPC (LoadLibraryA, handleThread, "malware.DLL")
  APC GENERATA PER UN'APPLICAZ.

Il malware, per nascondersi, cifra il suo contenuto e, in particolare, le strutture dati e le stringhe. Per farlo, esistono diversi meccanismi:

→ Effettuando lo XOR del contenuto del malware con un byte arbitrario (LA CHIAVE): questa tecnica però è sgamabile perché, al posto di tanti 0x00, nell'eseguibile compaiono tanti 0x BYTE_ARBITRARIO (≡ 0x CHIAVE)

→ Effettuando lo XOR del contenuto del malware ECCETTO i byte 0x00 e 0xCHIAVE (effettivam. cifrare 0xCHIAVE con la chiave dà luogo a 0x00, il che è fonte di ambiguità). → FEDELTÀ: l'analista può avere la possibilità di effettuare un semplice attacco brute-force sulla chiave: se conosco la lunghezza della chiave e una qualche stringa nota all'interno dell'header dell'exe, il gioco è fatto.

→ Effettuando operazioni di rotazione (ROR, ROL), o anche somme e sottrazioni, oppure delle varianti di base64. → BASE-64 È L'ENCODING USATO STD PER ALLEGARE FILE BINARI ALLA POSTA ELETTRONICA

21/12/2021

Codifica base-64:

È uno standard che prende un flusso di byte e lo converte in una sequenza di byte più lunga in cui ciascun byte, anziché prendere 256 valori possibili, ne prende solo 64 (quelli stampabili).

OVVERO
- LETTERE MAIUSCOLE (26)
- LETTERE MINUSCOLE (26)
- NUMERI (10)
- '+'
- '/'

⇝ $64 = 2^6$ ⇒ i byte "stampabili" sono composti da 6 ~~byte~~ bit.

Comunque sia, nella conversione alla codifica base-64, non è detto che otteniamo una seguen-

za di bit composta da un multiplo di 6 bit ⇒ la codifica base-64 prevede l'uso di un carattere di padding, che è "=".

↳ Il malware, per cifrare i dati (o il codice), usa delle varianti di base-64 (ricordare a base-64 così com'è è facilmente scamabile).
Le modifiche che possono essere attuate sono:

- Eliminare il carattere di padding (il che può far NON funzionare il programma).
- ~~Invertire~~ Cambiare l'ordine dell'alfabeto dei caratteri stampabili + cambiare proprio qualche carattere dell'alfabeto (e.g. sostituendo '+' con ';').

<u>Algoritmi crittografici standard:</u>
Il malware può utilizzare degli algoritmi come AES (che richiede una chiave segreta per decifrare i dati) oppure come MD5, SHA-256 (che sono basati sulle firme).
⇝ Non è detto che le chiavi segrete siano contenute nel malware stesso, altrimenti sarebbero ottenibili dall'analista.
↳ Più spesso il malware contatta un server remoto.

Il malware può sfruttare anche i protocolli a chiave pubblica / chiave privata.

⇝ Ricordiamo che il malware può utilizzare gli algoritmi crittografici non solo per cifrare se stesso, ma eventualmente anche per cifrare i file della macchina della vittima (in tal caso sarebbe un ransomware).
I ransomware usano per lo più i cifrari a blocchi (≡ meccanismi a chiave simmetrica), dove la chiave simmetrica può essere generata randomicamente.

Tuttavia, non tutti i malware usano la crittografia standard, la quale ha effettivam. degli svantaggi:
→ È richiesto tanto codice.
→ È facilmente rilevabile.

→ Vuoi o non vuoi, in qualche modo la chiave dovrebbe essere memorizzata nel codice (in particolare se viene usato un meccanismo a chiave simmetrica).

Per capire se il mw usa la crittografia standard, l'analista può:

- Vedere le stringhe del programma. → "OpenSSL 1.0.0.rc"
- Vedere le librerie e le API importate. → Crypt* CP* Cert*
- Usare dei tool che rilevano l'utilizzo della crittografia (e.g. KANAL ≡ Krypto ANAL yzer). (A VOLTE DÀ DEI FALSI POSITIVI)

→ Chi scrive il malware, se ha tanto tempo (e tanti soldi), può anche inventarsi delle varianti degli algoritmi crittografici standard affinché i tool non funzionino più e la vita dell'analista venga complicata.

Inoltre, il malware potrebbe anche usare + algoritmi in cascata: in tal caso, l'unica cosa da fare è analizzare (e decodificare) il codice macchina.

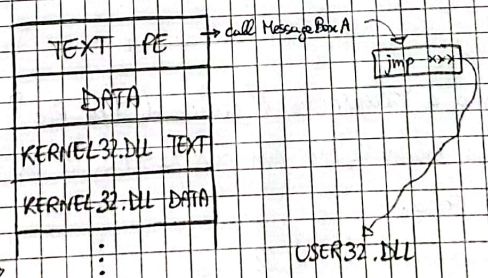
Per decodificare funzioni criptate, si possono seguire più approcci:

1) SELF-DECODING in cui si può far uso dei breakpoint.
2) Scrivere a mano una procedura di decodifica.
3) Effettuare il detour del malware alla funzione di decodifica (questo si può fare anche con uno script, in modo automatizzato).

Shellcode:

È un qualunque codice che può essere iniettato in sostituzione/all'interno di un processo in esecuzione.

Processo P:



Lo shellcode ha un problema: non può far riferimento a nessun indirizzo che può essere usato dal SO (perché non può far nulla di ciò che fa il SO) nel momento in cui inietta del codice.

→ CHIAMATA A FUNZIONE:

- call func
  - → VERSIONE ASSOLUTA: viene specificato l'indirizzo di func
  - → VERSIONE RELATIVA: viene specificato un offset rispetto a EIP
- jmp func2
  - → VERSIONE ASSOLUTA
  - → VERSIONE RELATIVA
- MOV EAX, var → SOLO VERSIONE ASSOLUTA ⇝ problema: nel momento in cui viene caricato il codice, non si conosce l'indirizzo assoluto della funzione/dell'eseguibile in questione.

Esiste un indirizzo preferito, che il SO non è obbligato a rispettare (⇝ il SO adotta particolari meccanismi che fanno uso di tabelle, ecc.)

Poiché lo shellcode non può far nulla che fa il SO, deve sfruttare altro, come il PIC (Position Independent Code).

Problemi dello shellcode:

→ Qual è l'indirizzo di base per il quale esegue? Non lo sa: la VirtualAllocEx, che alloca della memoria se il processo vittima, non ci dice a priori qual è l'indirizzo di memoria in questione.

→ Necessità di caricare librerie (API) e risolvere i riferimenti in memoria delle API di tali librerie. In quale indirizzo vengono caricate le librerie? Non lo shellcode non lo sa.

⇝ Come facciamo a riconoscere il codice di tipo PIC?

È come un codice normale ma non può far uso di indirizzi assoluti. Le strategie che può utilizzare sono:

- Invocare le istruzioni

```
call  to   ← Carica l'indirizzo corrente sullo stack
pop   ebx  ← Salva l'indirizzo corrente in EBX
                (togliendolo dallo stack)
```

. Usare le istruzioni macchina fstenv, fnstenv, che servono a fotografare il contesto (= insieme dei registri) del processore matematico (l'FPU).
↳ Floating point Unit

→ All'offset 0x0C c'è un campo del contesto chiamato instruction_pointer, che indica l'indirizzo dell'ultima istruzione FPU eseguita. (*)

↳ SI POSSONO INVOCARE LE SEGUENTI ISTRUZIONI:

```
fldz
fnstenv byte ptr [esp-12]
pop ebx
```

← ESECUZ. DI UNA QUALUNQUE ISTRUZ. FPU

← IN EBX ABBIAMO L'INDIRIZZO DELL'ISTRUZ. FPU

(*) Gli altri indirizzi assoluti delle altre istruzioni vengono ricavati di conseguenza, sfruttando gli indirizzi relativi rispetto all'indirizzo assoluto che, ad esempio, abbiamo memorizzato in EBX.

Anche le DLL hanno i loro indirizzi preferiti; raramente il SO ha necessità di rilocarle. Perciò, lo shellcode può fare delle previsioni su dove verranno caricate le DLL. Il problema è che SO differenti fanno uso di indirizzi di base diversi per gli eseguibili ⇒ se lo shellcode ha UN solo target ben preciso, allora OK.
↳ L'indirizzo di base può essere trovato tramite il debugger.

In generale, per risolvere gli indirizzi delle DLL (e delle loro API), si fa uso della coppia di funzioni LoadLibraryA e GetProcAddress: ma in quali indirizzi si trovano queste due funzioni? È un cane che si morde la coda...

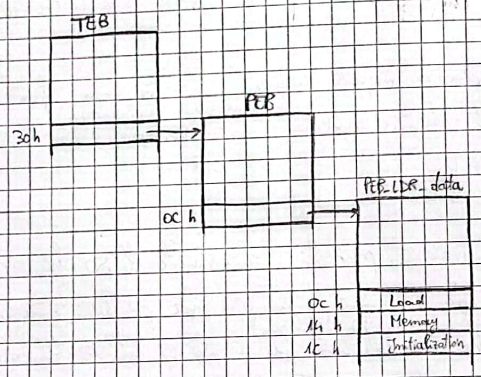
23/12/2021

→ Lo shellcode non è un eseguibile: quello che vi si può fare è un po' di analisi statica.
↳ NECESSITÀ DI INDOVINARE L'INDIRIZZO DELL'ENTRY POINT DELLO SHELLCODE.

→ Per eseguire lo shellcode e/o fare analisi dinamica, bisogna utilizzare lo SHELLCODE LAUNCHER, che è un'eseguibile che lancia lo shellcode.
↳ È COMODO AGGIUNGERE L'ARGOMENTO -bp QUANDO SI ESEGUE LO SHELLCODE LAUNCHER, PERCHÉ CONSENTE DI INSERIRE UN BREAKPOINT SUBITO PRIMA CHE VENGA LANCIATO EFFETTIVAM. LO SHELLCODE.

→ undocumented.ntinternals.net è un sito che prova a documentare ciò che la Microsoft non ha documentato.

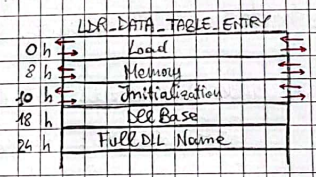
→ Lo shellcode, come dicevamo, se sa con certezza in quale indirizzo si trovano le API LoadLibraryA e GetProcAddress* (e, quindi, in quale indirizzo si trova la DLL Kernel32.DLL), ha risolto il problema degli indirizzamenti.

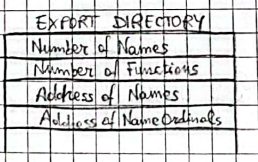


Questi sono puntatori a delle liste circolari in cui ogni elemento punta sia al precedente che al successivo

↳ In PEB_LDR_DATA abbiamo a vero il 1° elemento della 3 liste

Load punta a Load delle strutture adiacenti, memory punta a memory delle strutture adiacenti, ecc.





→ Sono indirizzi di nomi alle funzioni (stringhe), che sono ordinati alfabeticamente.

MessageBoxA, Sleep...

* In realtà, GetProcAddress potrebbe anche non servire perché la LoadLibraryA restituisce un handle a DLL-base (indirizzo base della DLL).

Sugli shellcode ci sono 2 operazioni che si fanno:

→ CODIFICA: spesso lo shellcode è una stringa e sfrutta funzioni come strcpy, strcmp, sprintf per vedere se si verifica qualche buffer overflow: se sì, il malware ha fatto bingo e può iniettare lo shellcode nel processo vittima sfruttando questo bug.

Il decoder è un modulo dello shellcode (composto da caratteri stampabili) che decodifica la parte codificata dello shellcode.

↳ che può essere arbitraria

→ NOP SLEDS:

NOP padding	decoder	encoded shellcode

Serve ad aumentare la probabilità di riuscire a iniettare lo shellcode nel processo vittima

↳ Utilizza i byte compresi tra 0x40 e 0x4f (0x90 non lo è stampabile).

11/01/2022

Malware moderno:

adopera in primo luogo delle tecniche di evasione sofisticate, per cercare di sfuggire a:

- Antivirus
- Firewall
- Intrusion Detection System (IDS)
- Intrusion Prevention System (IPS)

→ Un IDS raccoglie informazioni sul traffico di rete per scovare eventuali vettori di attacco.

→ Un IPS, in più rispetto all'IDS, ha un ruolo attivo (non solo di monitoraggio ma anche di prevenzione): quando trova qualcosa che non va nel traffico di rete, lo blocca.

Il malware moderno, per conseguire l'evasione, si appoggia sui protocolli di rete come HTTP, HTTPS, DNS.

↳ Di fatto, il traffico di questi 3 protocolli, tipicamente NON viene bloccato dai firewall. Inoltre, la mole di dati prodotti da questi protocolli è enorme ⇒ maggiore facilità per il malware di nascondersi.

Il malware può sfruttare questi protocolli in 2 modi:

1) Sfruttare degli aspetti secondari dei protocolli.

ESEMPIO: Nei pacchetti HTTP ∃ il campo USER AGENT (che indica il browser del client); questo campo non viene + usato dai server ⟹ il malware può anche dirne un comando malevolo nel campo user agent.

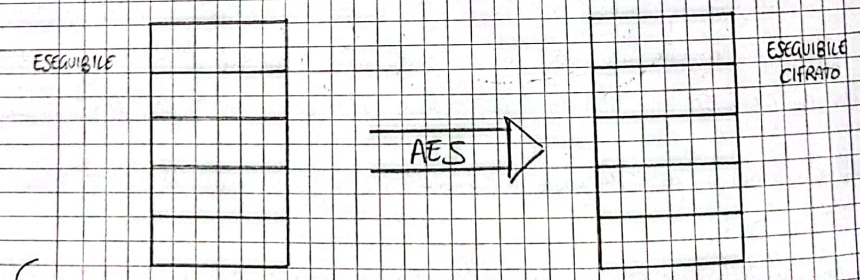
2) Inserire il codice malevolo nel payload dei messaggi HTTP (nelle pagine web nel caso di HTTP).

ESEMPIO: Un server malevolo, in un messaggio di risposta contenente codice HTML, può inserire un commento che codifica un comando, come <!--QU1xMjE=-->

ALTRO ESEMPIO: Anche un server DNS malevolo può sfruttare questo meccanismo, utilizzando i campi liberi dei messaggi DNS.

Un altro problema che il malware deve affrontare è che spessissimo le macchine infette si trovano dietro le NAT. Quello che fa il malware, è assegnare alle macchine infette (se sono + di una) un identificatore univoco (beacon), in modo tale che il server remoto possa riconoscere le diverse macchine.

Inoltre, il malware può sfruttare i packer in maniera particolarmente sofisticata, in modo da scoraggiare il + possibile l'attività di RCE.



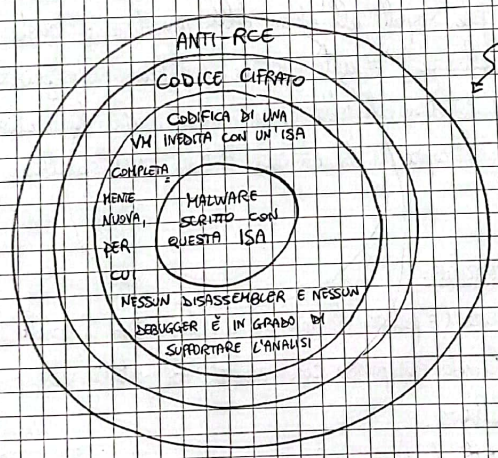
Il malware può decidere di NON salvare proprio la chiave di decodifica all'interno dell'eseguibile; in tal modo, per funzionare, cerca la chiave tramite la forza bruta.

Il grosso vantaggio è che, così, è possibile distribuire tantissime versioni di questo malware con altrettante chiavi di cifratura diverse (e, quindi, tante firme diverse)
⇒ facilita per il malware di evadere dagli antivirus basati sulle firme.

Alternativam., il malware può avere come target un'unica macchina fisica.
Così, sfruttando AES, può scegliersi una chiave di cifratura dipendente dal malware stesso e dalle CARATTERISTICHE FISICHE DELLA MACCHINA TARGET. In questo modo, i tool automatici chiaram. non funzionano più.

Un'altra cosa che il malware può fare è decifrare il codice originale un blocco per volta o una procedura per volta (NON tutto insieme).

La seguente altra tecnica è tanta roba:



UN ESEGUIBILE PUÒ ESSERE COMPOSTO DA TANTI STRATI CHE L'ANALISTA DOVREBBE AFFRONTARE UNO PER VOLTA A PARTIRE DA QUELLO ESTERNO
⇓
QUESTO RICHIEDE ANNI-UOMO DI LAVORO

<u>Possibile metodologia dell'analisi del malware:</u>

1) Detection del malware.
2) Isolamento ed estrazione del malware.
3) Basic static (code) analysis.

4) Basic dynamic ~~code~~ (behavior) analysis.
5) Advanced analysis.
6) Pattern recognition (≡ sintesi di ciò che si è capito dall'analisi).
7) Malware inoculation & remedy (≡ costruz. di sw che serve a "disinfettare" le macchine attaccate dal malware + meccanismi di prevenzione dal malware).

Per ciascun passo della metodologia, è bene specificare: OBIETTIVI, INPUT, PROCEDIMENTO, OUTPUT.

FASE DI DETECTION:
OBIETTIVI: stabilire la presenza e la natura del malware.
INPUT: libero.
PROCEDIMENTO: uso di antivirus.
OUTPUT: locazione dei file sospetti all'interno della macchina; hash del malware; categoria del malware; # antivirus che identificano l'eseguibile come malware.
→ In realtà, la fase di detection dovrebbe comprendere anche l'analisi dell'environment (→ il mw tenta in genere di capire in che ambiente si trova, - VM, sandbox, ... - e si comporta di conseguenza).

FASE DI ISOLAMENTO ED ESTRAZIONE:
OBIETTIVI: estrarre il malware (≡ portare il mw su un dispositivo in modo tale che sia innocuo; ovvero in formato compresso con uno zip ~~con~~ protetto con password).
INPUT: locazione del malware.
PROCEDIMENTO: uso di archivio protetto da password.
OUTPUT: archivio protetto.
→ Questa fase si complica se abbiamo a che fare con un rootkit; è possibile usare dei tool a questo scopo, come IceSword.

FASE DI BASIC STATIC ANALYSIS:

OBIETTIVI: trovare informazioni rapidamente.

INPUT: esemplare del malware che abbiamo estratto.

PROCEDIMENTO: analisi di stringhe, testata PE, metadati ed eventuali risorse dentro l'eseguib.

OUTPUT: le informazioni ottenute.

FASE DI BASIC DYNAMIC ANALYSIS:

OBIETTIVI: osservare il comportamento del malware.

INPUT: esemplare del malware.

PROCEDIMENTO: uso di sandbox (esempi di sandbox online: ThreatExpert, CWSandbox, Cuckoo);
uso di una propria VM con dei tool dedicati (Process Monitor, Process Explorer, ...).

OUTPUT: informaz. sul comportamento del malware.

FASE DI ADVANCED ANALYSIS:

OBIETTIVI: trovare informazioni tramite RCE.

INPUT: le informaz. ottenute precedentem. + esemplare del malware.

PROCEDIMENTO: uso di un disassembler interattivo e di un debugger.

OUTPUT: le informazioni ottenute.

FASE DI PATTERN RECOGNITION:

OBIETTIVI: ottenere una visione d'insieme del malware.

INPUT: le informaz. ottenute precedentemente.

PROCEDIMENTO: libero.

OUTPUT: IOC (indicatori di compromissione del malware); le vulnerabilità sfruttate dal malware.

FASE DI MALWARE INOCULATION & REMEDY:

OBIETTIVI: scrivere SW atto a prevenire e/o disinfettare le macchine dall'attacco del malware.