Malware Analysis

METODOLOGIA

1 - DETECTION
Identifico il malware, non ho info precedenti, cerco di riconoscerlo con antivirus.

2 - ISOLATION & EXTRACTION
Se voglio analizzare il malware, devo estrarlo in modo sicuro e non attivarlo.

3 - ANALISI STATICA DI BASE
Ottenziamo informazioni rapidamente, guardando stringhe, PE...

4 - ANALISI DINAMICA DI BASE
Vogliamo capire il malware, magari testandolo su sandbox o su nostra VM (con process monitor o regshot)

5 - ANALISI AVANZATA BASE/DINAMICA
Voglio più informazioni, uso disassembler e debugger per vedere il codice.

6 - PATTERN RECOGNITION
A questo punto, mi aspetto le maniogi e cerco di capire cosa il malware volesse fare.

7 - MALWARE INOCULATION & REMEDY
Come posso evitare che ciò riaccada? Come riparo ai suoi danni?

VULNERABILITA' IN RETE

Il malware usa protocolli semplici e difficili, come HTTP o DNS.
E' possibile intercettare artifatti locali del malware, su grande scala è molto più difficile.
- `NET` serve proprio in cui possiamo mettere dei controlli tra ciò che entra e ciò che esce. Ha senso se il malware viene dall'esterno, se è già dentro no.
- Firewalls/Proxies/Intrusion Detection Systems: ci permettono su rete locale, impostando regole locali (ad. aprire sito web ma intercettando dati) ma/ dice che tale indirizzo era ingiudicato da qualche parte). Non prevenire quindi quanto alcuni.
- Antivirus/Malware Scanners/IPS: Attività di prevenzione, come quando scarico un qualcosa che internet e mi dice che non è sicuro. Se lo fa con tutti i file, ha poco senso.

OFFUSCAMENTO DEL CODICE

SHELLCODE

Dato un processo generico in esecuzione, voglio inserirvi un codice che verrà eseguito ad un certo punto da questo processo. Stiamo parlando di shellcode. **Non va dove vorrà caricarsi**, non è assolo del SO. Può farci alcuni solo da indirizzi relativi per avere codice indipendente dalla posizione PIC. Gli indirizzi assoluti sono usati per puntare alle variabili (ex. mov [0x1234], ...)
In Windows, le off hanno indirizzo predefinito, non rilasci quasi mai Code so fra due dll a stesso indirizzo). Non è una codice rilocabile, quindi ho stessa pagina fisicha ma anche stessi indirizzi firmati (così non devo rilocare). Shell code usa PIC, ne usa API (indirizzo e prestabilito).

BUFFER OVERFLOW

Uno shellcode può forzare dal buffer-overflow sullo stack. L'input è di tipo testo dell'utente, è compito dello shellcode decodificarlo (riconoscere fine stringa, conversione ASCII, non sappiamo la situazione dello stack). Ciò che si fa è mettere NOP (0x90) prima del codice. Chiamo/ proiqua dè è assicurarsi di essere in mezzo a NOP (carattere, 2 F6 = 64 caratteri. A volte può richiedere padding di 100 caratteri, se lo ontanto, sto costeciando l'analisi.

Lo shellcode non è in binario puro (ha 0 e caratteri non stampabili) e quindi c'è fase di decodifica (produco solo caratteri stampabili), infine ASCII encoded decodifica e ci salto dentro.

COME SCRIVERE SHELLCODE

E' un file bin-raw-binary, NON ESEGUIBILE! Devo caricarlo in memoria. Uso una piccola fase iniziale che trasiamo operazioni FLD2 (per capire l'indirizzo a cui è caricato) e FSTENV (dato coprocessore mathematico). Quest'ultima ci serve perché con l'indirizzo dell'ultima operazione floating point, spiazzandoci di 12, arriviamo alla cima dello stack. Con CALL + POP ritorno l'indirizzo e ce lo caricato lo shellcode.

PROBLEMA DELL' INDIRIZZO A PRIORI

Molto spesso ci servono LoadLibrary e GetProcAddress, di cui dobbiamo conoscere gli indirizzi. In particolare, quasi tutti i processi usano kernelbase.dll (oggiuda a kernel32.dll). L'idea è quindi quella di ottenere proprio questa DLL, che contiene le due API.

1 - COME TROVARE KERNELBASE.DLL

Ogni thread ha Thread Environment Block (TEB) raggiungibile da FS:[0] all'offset 0x30 (x ha puntatore alla PEB (Process Environment Block). Nella PEB, ad offset 0x0C c'è puntatore a PEB_LDR_DATA. Essa contiene una serie di campi per descrivere la lista delle DLL caricate nel processo, in tre liste doppie circolari:
- InLoadOrderLists: in base a quando l'ho caricata.
- InMemoryOrderLists: in base alla memoria.
- InInitializationOrderModuleLists: in base all'inizializzazione (+ dei caricamenti). Quest'ultima è preferita, in quanto l'ordine varia prima NTDLL.DLL e poi kernelbase.dll. Ciò mi permette di ottenere l'indirizzo di kernelbase.dll. Ciò che ci interessa è il campo Flink, che ci dà l'indirizzo del prossimo elemento di questa lista. Ciò che ci può servire è quindi l'indirizzo dell'entry, ma off offset di InitializeOrder). Con questo, sapotoranado, troviamo Kernelbase.dll.

2- TROVARE INDIRIZZO LOADLIBRARY API

- Nella testata, Nter Exe header, + 1Ch dove 1Ch è un **Relative Virtual Address**, ovvero il prossimo header e trova ad un indirizzo base dell'intvalle .dll. Ciò mi dà l'indirizzo di Export Directory.
- Nella sezione File Header, troviamo ExportDirectory, altro RVA: ciò perché abbiamo una struttura dati contenente AddressOfFunctions, AddressOfNames, AddressOfNameOrdinals dalle API. Fare dll base + RVAExportDirectory + trovare tabella exports.
- Noi conosciamo AddressOfFunctions, che ad un indice n ci dà l'RVA di nostro interesse. Noi non conosciamo questo n, quindi partiamo da AddressOfNames (il nome lo sappiamo) me questi ritorna l'indice m nel vettore di nomi (non indirizzo). Con AddressOfNameOrdinals, fornendo l'indice m, troviamo n. Diamo poi n ad AddressOfFunctions e abbiamo fatto.

OFFUSCAMENTO DELLE STRINGHE

- **XOR**: si sceglie una chiave, e si fa XOR byte a byte. In particolare, non vogliamo troppo 0 visibili (facile decifrarlo). Limitiamo ciò non facendo XOR ke 00 == 0 (ke 00 == 00) perché produrrebbero 0. Esistono anche **shift/ rotazioni**.
- **BASE64**: Abbiamo rappresentazione binaria (ogni rappresentazione 6 bit), raggruppiamo a 3 byte (24 bit) e li dividiamo in 4 caratteri. Poiché ho 6 possibilità in decimale e 64 caratteri, ho corrispondenza. Se non ci sono 3 caratteri. A volte può richiedere padding di 100 caratteri, se lo ontanto, sto costeciando l'analisi.
- **Funzioni crittografiche**: Usiamo ad esempio AES con costanti modificate.

Misure Anti Virtual Machine

CPUID

- Ritorna stringhe differenti (associate al tipo di processore) se siamo VM o meno. Con QEMU non da problemi.

TIPI DI UTILIZZO

- Se lo usiamo con eax = 1, ritorna molti flag associati alla VM. Ad esempio, il 31 esimo bit di "ecx", viene usato per dire se si usa una VM. (`1= true` ; `0= false`)
- Se usiamo codice "0x4F000000", ritorna il costruttore della VM nei registri ecx, ebx, edx. Easy FIX mettiamo NOP.

CHIAVI DI REGISTRO - regedit

Cerchiamo chiavi associate alla VM. In quali lecciamo tracce (ad esempio per cartelle condivisa).

PROCESSI IN ESECUZIONE

Possiamo cercare servizi contenenti stringhe associate alla VM, o file o driver.

ISTRUZIONI VLNERABILI

Tabelle come SGDT, SIDDT, SLDT, CPUID, STR, SMSW varcano per manipolare strutture dati.
Molte VM modificano il formato delle tabelle puntate.

TECNICA PILL

Per capire se l'esecutione è su VM o meno.
- NO PILL: basata su SIDT
- NO PILL: basata su SGDT e SLDT. Puntiamo al loro formato.

PROCESSORI VIRTUALI

Hanno cose in più rispetto a quelli normali, ad esempio permettono di creare collegamento tra sistema GUEST e HV. Un esempio è Guest Additions.

OBIETTIVI E MECCANISMI SFRUTTATI DAL MALWARE

PERSISTENZA

- Uso di ciclosi: Il malware vuole continuare ad avere il controllo della macchina. Il malware si rlancia insieme al PC, grazie a chiavi di Windows, usate per memorizzare la configurazione. (Utilizza CurrentVersion\Run). Le vedo con regedit. Con Autoruns vedo i programmi allo startup.
- Uso delle dll: Le dll sono come file eseguibili, ma con più entry point, uno per API. Con il meccanismo AppInit DLL, vengono caricate le dll facenti uso di user32.dll (molto comunici) posso mettervi malware!
- Windows Hooks: Ad alcuni come login, logoff, startup, viene caricata una dll, che rimane persistente dopo la prima chiamata.
- Windows Services: Come i demoni Unix, partono alla startup, creando una configurazione memorizzata in ControlSet (ce n'è uno Current, e vecchie configurazioni per il ripristino). Creare un proiqua! eqoiqua! Ed il malware non ha un eseguibile, e appoggia a SVCHOST.EXE, **questa lipoiqua include anche svgestsvc**.
- Scheduled System Binaries: si usa EXPLORER.EXE (parte a startup, per vedere i file), sempre attivo. Il malware può modificarlo tramite **DETOUR**, ovvero far eseguire normalmente files ad un certo punto, cambiare una parte di esecuzione, e tornare sul flusso principale. L'accortezza sta nel creare dei «Dummies» nel codice, perché la rilocazione in memoria potrebbe creare problemi. Nel codice detour avremo bisogno di malware.dll da caricare con LoadLibrary, spiazzata di un certo offset da Kernel32.dll (basta creare un «Dummies Lora call) prima di fare questa operazione per caricare la libreria malevola, per poi tornare sul flusso originale. Alternativa firma programma, **include anche scrubbing privilegi**.
- DLL load order Hijacking: si basa su come EXPLORER.EXE trovi le librarie. Cerca in punti precisi, ovvero cartella applicazione, cartella corrente (dove lancio), cartella di sistema, path. Se le mia dll malevola si chiama come una ufficiale e la precede in questa ricerca, verrà usata lei, non ci sono controlli. Esiste una lista di **KnownDLL**, che non funzionano (ciò), se lo ne trovo una dll esterna e tale lista, ma presente in explorer.exe, fin riavvio.

SEGNETEZZA

Il malware vuole nascondere l'esecuzione all'utente, per non farsi scoprire ad inizio esecuzione. L'idea principale è usare i **rootkit**, che si installano nel sistema per nascondere a loro volta un malware interno.

- Rootkit user mode: sfruttano Import Address Table Hooking: Un eseguibile può usare API Messaggiflo. Scrivere parte di una dll esterna linkata dinamicamente, NetEseguibile v1.4.7 (ciò ci eseguo jump nella dll e questa API). Il malware può solo bloccare questo salto a questa API, sefiando sul suo codice. Alternativa: Inline Hooking: Ogni volta che qualcuno chiama la API, Kernel32.dll legge, Kernel. Cambia la firma ufficiale. Sostituzionamo IRT salva a ciò che dice lo, e poi lo salto ad un'altra. Ciò è più difficile.
- Inline Hooking: Faccio detour modificando la dll. Quando ritorno sul flusso originale, cancello le prove. Posso fare detour in memoria. (fatto 2).

EVASIONE

Il software può presentare comportamenti anomali.

- Dll Injection: crediamo che carica la dll malevola. Devo, per iniziare, prendere il PID del processo vittima esistente ed ottenerne l'handle (con OpenProcess): per controllarlo esternamente. Il mio processo può allocare memoria al processo vittima (con VirtualAlloc) per scrivere codice) e poi scriverci (con WriteProcessMemory), e cambiare il suo flusso con CreateRemoteThread. Opero in memoria, non mi scopronno. Attacco sempre esplorer.exe che ha tanti ridere per le dll.
- Code Injection: come prima, ma senza dll, solo detour, il codice viene eseguito nel processo. Fattibile se ho problemi di privilegi (processo meno importante modifica pagine di un processo più importante).
- Process Replacement: enhance SVCHOST.EXE, le tecnica mi permette di lanciare un processo per poi sostituirlo con un altro grazie a CreateProcess con flag SUSPENDED (crea ma non lancia). Poi devo buttare via tutto ciò che il SO ha fatto per creare il processo (lascio solo i privilegi), usando la ZwUnmapViewOfSection. Sto lavorando in un guscio vuoto.
- Hook Injection: Intercettiamo le deviazioni (hook) dei messaggi dei processi. Utente crea evento che richiama il SO, ciò comporta un messaggio che notifica dll per la gestione. Tale messaggio viene passato al thread dell'applicazione che carica tale dll. ClassicInjection: Hook Injection, ma con messaggio ad hoc.
  - Remote Hook (kernel level): lego che installa Thook deve rimanere in esecuzione.
  - Remote Hook (user level): lego che installa Thook deve rimanere in esecuzione.
  - Remote Hook (high level) malware che lo installa può chiudersi, dll carcata dove viene richiamato l'evento. Usato con Keylogger. Ha lo esempio: SetWindowsHookEx("WcpDHook, functionalityLegacy, HandleEventProcedure, ChiRiceveHook)
- Asynchronous Procedure Call: Facciamo eseguire routine quando un processo è idle, tramite WaitForSingleObjectEX se presente (simile ai semafori). Il malware passa cose che immobilie fare al processo in questione, che non avrà problemi a farlo.

PACKER

INTRODUZIONE

Nasciono dalla necessità di ottimizzare lo spazio. Oggi usati dai malware per difesa. Dato un file eseguibile, abbiamo PE, testo,dati etc.
L'applicazione di un packer cambia tale struttura, in quanto rimane PE, il testo diventa stub, e c'è solo un'altra sezione data. Lo stub legge i dati da data per ricostruire in memoria l'immagine originale. Lo stub ha cose diverse dal programma originale, e tramite operazioni di scrambling lo stub viene compilato per ostacolare l'analisi.

COSA FA LO STUB

- Esegua unpack dell'eseguibile originale in RAM. Usa un proprio algoritmo.
- Risolve gli imports, ovvero quali API di quali DLL vengono usate. Esiste una IMPORT ADDRESS TABLE che ce lo dice, gestita dal SO. Lo stub sistema la IAT, in tre modi:
  - Mantiene la IAT originale, facile ma non protegge
  - Si mantiene la IAT ma, per ogni DLL, si lascia una sola API, poiché l'ordine è prestabilito. Lo stub sa quali sono le altre API da reintrodurre. Quando c'è richiamo a DLL, ci saltiamo dentro.
  - Non usiamo il SO, quindi reinserisce la IAT, bensì le API **LoadLibrary** (carica le dll nel processo) e **GetProcAddress** (passiamo nome API, torna indirizzo). Quindi lo stub carica le dll necessarie e le comunica ad un wrapper, che varia chiamata dal programma originale per indirizzarlo verso queste API.
- Salta all'ORIGINAL ENTRY POINT. Non è banale capire dove si trova.

REVERSING

Dobbiamo partire dallo stub, che non è il programma originale, quindi sarebbe buona cosa non perderci troppo tempo. Con tool di analisi statica e dinamica, dovremmo trovare l'OEP, per capire quando lo stub termina.
Cosiamo l'immagine creata dallo STUB, che è un file non eseguibile dentro OLMAP, in quanto non ha IAT. Stub opera in memoria, sistema gli indirizzi del wrapper, non corregge la IAT. Importiamo la IAT alla fine.

TROVARE OEP

- Cercchiamo i CODE JUMP: Stub piccolo, lecito sia dato e in pagina di memoria allocata dinamicamente sull'heap. Ciò vi aspettiamo un salto lontano all'OEP nell'heap. Usiamo disassembler.
- Dump: Quando il programma originale è in esecuzione, lo stub non è più in memoria, quindi posso fare dump della memoria. Ma impacchettando, a quindi salva tutto sullo stack. Quando avremo settato al programma originale, piazzatemi lo stack con PUSHAD. Quindi prima di questa operazione mi aspetto un lungo jump.
- Quindi dalla memoria: Se siamo riusciti a trovare l'OEP prima, il programma dovrebbe partire, quindi ha rischio la IAT.
- Visualizzazione 1 File (padre): non separato la dimensione del file originale, prendere più pagine del necessario.
- Visualizzazione 2 Con Importlist (polverissimo), ma richiede OEP e processo attivo. Se corretto, ci permettono possiamo vedere se la ricrea correttamente.

PACKER PIU' RECENTI

- In versioni recenti del packer la chiave non è inclusa nel packer stesso, bensì potrebbe usare forza brute (può farlo anche l'analista, ma deve sapere che la chiave non è interna) o appoggiarsi ad un server con chiave remota (quindi dobbiamo intercettarla).
- Il packer potrebbe decodificare il malware a blocchi, non tutto insieme.
- Il malware potrebbe essere scritto in un linguaggio comprensibile solo ad una macchina virtuale creata appositamente per eseguirlo. Ciò introduce anche le **sandbox**, poco scalabili (ok se controllo pochi malware), e il malware non deve capire il vero dentro.

COME RICONOSCERE FILE PACKED

Ciò che possiamo fare è:
- Vedere i nomi delle sezioni: con il programma UPX abbiamo sezioni UPX0 (stub), UPX1 e UPX2 (data).
- API usate: Tipicamente VirtualProtect (diritti pagina memoria), LoadLibrary e GetProcAddress.
- Differenza Virtual Size e Raw Size: se molto diverse, è sospetto.
- OAT: se GHidra mostra molti OAT, vuol dire che non comprende le chiamate.
- Entropia: posso analizzarla, ma se non so quale sono quelle original.
- Entropia 7: massima se il byte precedente non mi dà alcuna info su quello successivo. La compressione alta l'entropia (valore circa 7, senza packer sui 6), inglie cose ridondanti.

PRIVILEGI

Il malware parte a livello user, e poi prende privilegi superiori.
C'è user, admin e NT_AUTHORITY_SYSTEM (controlla tutta la macchina nel dominio). Windows assegna ad ogni processo una capacità elementare (Editto di fare operazioni). Ogni processo ha un token associato, che contiene dono dei token ricevuto con API. Ciò ciò che si può fare è usare tool automatici per cercare vulnerabilità, come Metasploit o exesing.

Malware vs Disassembler

Problema con i jump

Classifichiamo tre tipi di jump:
- JMP: incondizionate, il flusso passa ad etichetta specifica.
- JZ: ad ultima operazione ha ritornato zero, salta.
- NZ: ad ultima operazione NON ha ritornato 0, altrno salta.

Problema

Ho un jz 0; e quindi so cosa bifforcazione posso andare all'etichetta 0 (se salto), oppure a q (se non salto, subito dopo). Ma il disassembler non sa sempre q. Ci fa metter sullo stack e ne prescovpa dopo.

Se in q ci fosse jnz 10?

Come prima, mettiamo jnz 0 sullo stack, e proseguiamo con c convero l'istruzione subito dopo.
Il problema si crea, jz 0 e jnz 0 = jump 0; quindi non dovrei mai disassembrare quello che sta ad i. però GHidra non lo capisce (vede due salti incondizionati) e processa male le istruzioni!

Questo vuol dire che potremmo salvare nel mezzo di un'istruzione, facendole perdere di significato. Ad esempio, vdi c6 04 24 0C ad, so saltiamo di +1, leggiamo "x0" come "call", e il resto come indirizzo. Ma questa call, così interpretata, non la eseguiremo mai.

Per risolvere, bisogna disattivare il codice machina

Salto nel mezzo di un'istruzione

Per il disassembler, uno stesso byte non può essere associato a due istruzioni macchina diverse. Per il processore è...
Ne è un esempio 0B FF C0 48, dove:
- 0B FF: OR EDI, EAX. Ciò rispetto all'instruction pointer (ovvero indirizzo prossima istruzione, ovvero C0), quindi da C0 nominano ad FF.
- FF C0 è "INC EAX", però GHidra non accetta che FF faccia parte di due istruzioni. Ciò ci ritorna un errore.
- 48 è "DEC EAX".

Failure flusso esecuzione

```
    call 10
10:  add [esp], 5
     ret
10+5: ...
```

La particolarità è che l'indirizzo di ritorno della call, è 40 ulteriori, addirittura 4 byte, sommiamo 5 così torniamo su ret. A quel punto la return porta esattamente ad eseguire l'istruzione dopo retn.
Per GHidra la funzione finisce con return, ma non sicamo alterando la visione d'insieme, nascondendogli altra call.

Divisione per 0

Sappiamo che FS:[0] deve puntare all'handler più esterno, che chiamiamo ID, da noi introdotto sullo stack con una PUSH. Quando ci sarà un'eccezione, il processore ci salterà all'indirizzo di ID, ma prima di farlo, ci mette in EAX l'indirizzo di ID (che abbiamo messo il nuovo handler). Questo nuovo gestore è pardi programma per non ritornare mai il controllo. E' presente anche un handler di sistema che punta ad ID.
EAX punta a tale handler di sistema. Se dereferenziamo due volte EAX, andiamo prima su ID, poi sull'handler più esterno.

Cosa fa ottenuto?

Facendo quesiti "sello" non ritornieimo più sul flusso di esecuzione principale, l'handler non ritorna mai e quindi il codice non continua l'esecuzione.

STRUCTURED EXCEPTION HANDLING SEH

Cos'è?

Estensione del linguaggio, offerta da Windows. Usato per contesti low level di tale SO (driver, o caratteristiche di Windows).

Come è fatto?

Mantenimo sullo stack una lista dinamica di gestori annidati) con un campo prev (indirizzo precedente) e un campo handler.
Sullo stack abbiamo anche 2 byte di annidamento. Se non trovo handler adatto, si usa il default. La lista è per ogni thread, in quanto ogni thread ha il suo stack.

Dove si trova la testa?

Nel Thread Environment Block, ad offset 0 abbiamo puntner al primo elemento. Ci arriva con **FS:[0]**.

Come crea un nuovo Handler?

```
PUSH EscapeLoadingerror ; EscapeFrame, dd ESP_exception, dd ignore
PUSH FS:[0]
MOV FS:[0],ESP
```

Lista gestori

Tipi di disassemblatori

Lineari

Non comprende il codice, ne assicia 2 byte ad una istruzione, non considera eventuali salti nel mezzo.

Interattivi

Segue la semantica delle istruzioni, si rialinea nel jump. Si interfaccia con l'analista.

Differenze nel codice

Bit

32 bit o 64 bit.

Architetture

Intel: <destinazione, sorgente>
AT&T: <sorgente, destinazione>

Endianess

- Big Endian: byte più significativo va a sinistra in **memoria**.
- Little Endian: byte più significativo va a destra in **memoria**.

Rimozione degli argomenti

- stdcall: chi viene chiamato pulisce lo stack.
- cdecl: chi chiama pulisce lo stack.

Ottenimento del codice macchina

Codice sorgente

singola unità di compilazione, posso averne 11

Precompilazione

sostituisce le #include e #define, non risolve.

Compilatore

Ci fornisce codice assembly/oggetto. Non ha ancora tutto per eseguire.

Assemblatore

Traduzione in linguaggio macchina parziale, non ancora eseguibile.

Linker

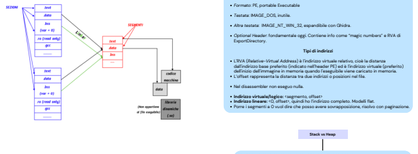
Combina più file assembly e le librarie di sistema, risolvendo i riferimenti e generando l'eseguibile.

Libreria statiche: Semplicemente, a compile time, i richiami della funzioni con l'indirizzo della procedura. E' quindi già eseguibile.

Librerie dinamiche: Codice non incluso nell'eseguibile, ma caricato dinamico a runtime. Linking col SO.

File oggetto vs Eseguibile

Associazione tra SECIONI e SEGMENTI. Processo a diverso attiva del programma.



Caratteristiche del codice

File eseguibili 32 bit Windows

- Formato PE, portable Executable
- Testata: IMAGE_DOS, inutile.
- Altra testata: IMAGE_NT_WIN_32, espandibile con GHidra.
- Optional Header: fondamentale oggi. Contiene info come "magic numbers" e RVA di ExportDirectory.

Tipi di indirizzi

- L'RVA (Relative-Virtual Address) è l'indirizzo virtuale relativo, cioè la distanza dall'indirizzo base preferito (indicato nell'header PE) ed il (indirizzo virtuale (preferito) definito dall'immagine in memoria quando l'eseguibile viene caricato in memoria.
- L'offset rappresenta la distanza tra due indirizzi o posizioni nel file.
- Nel disassembler non esegue nulla.
- **Indirizzo virtuale/logico**: 1segmento, offset!
- **Indirizzo fisico**: 1 numero, ma è il processore completo. Modelli flat.
- Porte i segmenti a 0 così dire che posso avere sovrapposizione, rischio con paginazione.

Tipi di disassemblatori

ANALISI STATICA (malware morto)

- **BASE**: cose contenute il file, quindi si cerca hash, stringhe, oppure PE header (PE-Studio)
- **AVANZATA**: vado assembler, interazione con utente (GHidra).

ANALISI DINAMICA (malware vivo)

- **BASE**: monitoraggio, ad esempio wireShark.
- **AVANZATA**: uso debugger, come OllyDBG.

Ovviamente, più entro nei dettagli (avanzata), più il processo è lungo.

Tools utili

PROCESS EXPLORER:

- **Problema**: un malware può lanciarsi come eseguibile ufficiale, evvitarlo e scoltrarlo.
- **Cosa fa**: trattiamo chi lancia cosa, con PID, sino 1 malware si nascondono tra i processi di sistema, lanciati dai services.exe. Process Explorer vede la firma rispetto all'originale, e le differenze tra originale.

Process Monitor

Monitora il sistema durante l'esecuzione. Registra le API e chiavi.

Regshot

Esegue due fotografie, pre e post malware.

Wireshark

Osserva il flusso dei dati a livello rete.

ApateDNS

Attiro le richieste IP del malware, magari facendolo comunicare con una macchina da me controllata.

ApiMonitor

Alternativa di ProcessMonitor, utile con API low-level, per capire cose fa l'applicazione.

OllyDBG

- Fa parte dell'analisi **grey box**, ovvero eseguo il malware controllata.
- Formisce finestre sul disassemblaro, registri, dati e stack
- Un debugger lavora in Kernel Mode o User Mode, a livello Assembly o livello Source code.

Breakpoint

Permette di bloccare l'esecuzione a tempo opportuno in un punto specifico. Abbiamo:

- Breakpoint in esecuzione: basati su INT3 o 0xcc. Ad esempio divido per 0.
- Breakpoint quando accedo a locazioni in memoria: (+prm e page fault (scrivo su pagina non scrivibile).
- Breakpoint software a scelta: 0 int
- Breakpoint hardware: Trovviamo su registri che generano eccezione fino. Il malware si difende poco.

Il trap flag non è un breakpoint, in quanto serve per eseguire "passo passo", in cui dopo ogni istruzione macchina interviene il debugger, e l'esecuzione si ferma.

Tipologie di analisi

Come analizzare

- **PE-studio** per vedere entry point, creazione, sezioni, eventuale offuscamento (vedendo virtual size e raw size).

2- Analisi di base dinamica

- Usiamo Process Explorer, Process Monitor.
- Passiamo a GHidra se necessario, magari per cercare stringhe.
- Regshot pre e post lancio malware.

NB: Ricerca del main

Se malware è grafico, cerchiamo il ciclo del messaggi, ahimenti una funzione con parametri compatibili con argc, argv.

Malware vs Debugger

Detection

Il malware si rende conto di essere sotto debug, allora il suo comportamento. Può semplicemente non funzionare, o depistare l'analisi. E' diretto.

- **IsDebuggerPresent()**: API usata per vedere la presenza del debugger durante l'esecuzione del processo che esegue l'API. Possiamo patcharla con disassembler.
- **CheckRemoteDebuggerPresent()**: Simile a prima, ma qui posso passare handle di altri processi.
- **NtQueryInformationProcess**: Use ProcessDebugPort che ritorna 1 se il debugger è presente.
  - Usiamo NtSetInformationProcess è cancella a runtime, con ProcessBreakOnTermination per prendere l'indirizzo della funzione con pathcheckup.
- **OutputDebugString()**: Se non c'è debug, ritorna un errore diverso da un errore precedentemente settato con setLastError e preso con getLastError. Se errore vecchio e nuovo sono uguali, c'è debugger.

Identifying

Il malware non si rende presente contro di essere sotto debugger, ma note dei pattern tipici. E' indiretto.

- **Ricerca strici/cartelle**: Controllarmo tracce lasciate dal debugger, come CurrentVersion\Uninstallwhog configurate sa c'è debugger.
- **Analisi finestre aperte**: Usiamo FindWindowA per trovare handler con nome delle finestre, magari il nome della finestra è associata al debugger.
- **Scan della memoria**: Cerchiamo in memoria le operazioni, INT3/ 0xcc, ad esempio chiudendo "REPNE" per trovare un byte nell'area di memoria.
- **Timing Checks**: Tramite rdtsc, vediamo il tempo di esecuzione, se troppo alto, dovremmo far terminare l'esecuzione per ripartirvi.

Interference

Non vedo prima e vedere se c'è debugger: questa interferenza ci sono a priori. L'idea è che il malware mette INT3 per prendere il controllo prima che lo faccia il programmatore. Ad esempio, un programmatore lo mette in entry point, ma prima c'è la modalizzazione del TLS, ke cui posso lavorarci.

Il malware può anche generare eccezioni stupide (adoperabili) o meno stupide (INT3). OllyDBG capisce quelle che ha messo lui.

EXCEPTION vs INTERRUPT

- **Exception**: Evento sincrono, dipende dall'esecuzione delle istruzioni(divisioni per 0, page fault). Posso ignorarle o catturarle, serve il processo giusto.
- **Interrupt**: Evento asincrono, dipende da dispositivi esterni.

Exploiting Vulnerabilities

Il debugger non è esente da bug.

- **RVA**: Possiamo avere size di 64 cattore, se lo modifichiamo con valore più grande, Windows non ha problemi (oltre 8 non ce). OllyDBG si sconcerta: erre di codice.
- **Raw vs Virtual size**: Non sempre coincidono, OllyDBG oggi prende il minimo tra la dimensione del file su disco o in memoria. Prima, se Raw size > Virtual Size, avevo problemi.
- **Strings mal processate**: Ruby exploits OllyDbg/Immug: mando una stringa + 5xffxfdx create perdite OllyDBG crede siano pulitelori valid, ma ad un certo punto la stringa termina, ci sarà un puntatore a 0 che casa segfault.

Malware vs Debugger

BeingDebugged

La struttura dati Process Environment Block è usata dal S.O. per memorizzare informazioni rilevanti per il processo. Ogni processo ha la propria PEB.
Essa include un parametro BeingDebugged.

Come ci arriva?

Windows usa indirizzi linear. Ogni thread (di un processo) ha struttura dati ad indirizzo 1fd0ad 0h. Ad offset 30h c'è puntatore a PEB (unica per processo!) Arrivati alla PEB, ci spostiamo 0x02h per raggiungere il flag.

```
MOV EAX, FS:[0x30]
MOV EAX, dword ptr [EAX + 0x02]
```

Heap

Attraverso 0x18h della PEB, c'è Processheap per gestire la parte di memoria per l'allocazione dinamica.
La gestione cambia in presenza di debugger, toccando **Flags** e **ForceFlags**:
- se D, non c'è debugger.
- se 0 D, c'è debugger.

I problema è che cambiamo posizione (offset) in base alla versione di Windows.

	WinXP	Win10
ForceFlags	0x10	0x44
Flags	0x0c	0x40