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 alterarlo.

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

4 - ANALISI DINAMICA DI BASE
Vogliamo capire il malware, magari lasciandolo 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 spremo le meningi 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 diffusi, come HTTPS o DNS.
È possibile innescare attività local del malware, su grande scala è molto più difficile.

- **ARP**: serve per sapere mac dell'indirizzo IP, mentre dei controlli ma ciò che entra e ciò che esce. Ha senso se il malware opera dentro la stessa LAN.
- **Intrusion Detection System IDS**: Osserva traffico su rete locale, impostando regole local (es: aprire sito web mai interrogato da DNS root dire che tale indirizzo era ingiobato da qualche parte). Non prevalente su grandi reti.
- **Intrusion Prevention System IPS**: Attività di prevenzione, come quando scarico un qualcosa da internet e mi dice che non è sicuro. Se lo fa con tutti i file, ha poco senso.

OFFUSCAMENTI E SHELLCODE

OFFUSCAMENTO DEL CODICE

SHELLCODE

Dato un processo generico in esecuzione, voglio inserirgli del codice che verrà eseguita ad un certo punto da questo processo. Stiamo parlando di shellcode. **Non ne deve verrà caricato**, non è accetto dal SO. Può farsi accadere solo da indirizzi relativi, per avere codice indipendente dalla posizione PIC. Gli indirizzi assoluti sono usati per puntare alle variabili (ex: mov eax,[ret]).
In Windows, le all'hanno indirizzo predefinito, non rilocc quasi mai (solo se ho due dll a stesso indirizzo). Non è usa-codice che funziona in qualsiasi pagina fisiche ma anche stesso indirizzo linear (cosa non deve rilocare). Shell code usa PIC, ed usa API Dinamiche e preenabili.

USE CASE: BUFFER OVERFLOW

Uno shellcode può forzare del buffer-overflow sullo stack. L'input è di tipo testo dell'utente, a compito dello shellcode decodificarlo (riconoscere fine stringa, conversione ASCII, non sappiamo la situazione dello stack). Ciò che si fa è mettere nello stack (non SO, poiché non sfarrapolito) all'inizio per realizzare padding, cosa serve più scuri di andare all'inizio dello shellcode.
Lo shellcode non è in binario puro (ha 0 e caratteri non stampabili) e quindi c'è la fase di decodifica dei caratteri (produco solo caratteri stampabili), infine ASCII-encoded shellcode, con cui il decoder prende la lista di caratteri stampabili e costruisce il vero shellcode, per poi saltarci dentro.

COME SCRIVERE SHELLCODE

E' un Stack-ter rilevitory, NON ESEGUIBILE (non ha header). Ghidra capisce poco, ma trovamo operazioni FLOD (salva valore in stack, operazioni su stack, operazioni su FLOD). Ciò che si fa è mettere in memoria, tramite operazioni su stack, l'ultima operazione floating point usata). Quest'ultima si serve perché con l'indirizzo dell'ultima operazione floating point, spiazzandosi di 12 (ESP-12), arriviamo alla cima dello stack. Con CALL + POP ritorno l'indirizzo a cui è caricato lo shellcode.

OFFUSCAMENTO DELLE STRINGHE

- **XOR**: si scegliе una chiave, e si fa XOR byte a byte. In particolare, non vogliamo troppo 0 (XOR 0 non cambia nulla), quindi si sceglie chiave diversa da 0 (XOR 00 ok) — C++ ok (C++ SQ, perché produrrebbero 0). Esistono anche **shift, rotazioni**.
- **BASE64**: Abbiamo rappresentazione binaria (ogni rappresentazione 8 bit), raggruppiamo a blocchi di 6 perché ci sono 64 caratteri ASCII diversi. Canovi gruppo da 6 associo valore decimale e poi codificato. 2^6 = 64 caratteri. A volte può richiedere padding di 1/2 caratteri, se lo ometto, sto costacciando l'analisi.
- **Funzioni crittografiche**: Usiamo ad esempio AES con costanti modificate.

Misure ANTI Virtual Machine

CPUID

- Ritorna stringhe differenti (associate ai tipi di processore) se usiamo VM o meno. Con QEMU non ok problemi

TPI DI UTILIZZO

- Se lo usiamo con eax = 1 ritorna informazioni sul processore VM. Ad esempio, il 31-esimo bit di "ecx", viene usato per dire se si è sia una VM (= 0) o no (= 0)
- Se usiamo codice "0xf000000000", ritorna il costruttore della VM nei registri ecx, ebx, edx. Easy fix: rewriamo NOP.

CHIAVI DI REGISTRO - regedit

Cerchiamo chiavi associate alla VM, le quali lasciano tracce (ad esempio per cartelle condivise).

PROCESSI IN ESECUZIONE

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

ISTRUZIONI VULNERABILI

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

TECNICA PILL

Per capire se l'esecuzione è su VM o meno:
- MEO 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 collegamenti tra sistema OUEST e VM. Un esempio è Quest Additions.

OBIETTIVI E MECCANISMI SFRUTTATI DAL MALWARE

PERSISTENZA

- Uso di «Hook». Il malware vuole continuare ad avere il controllo della macchina dopo l'infezione. Il malware si riavvia insieme al PC, grazie a chiavi di Windows, usate per memorizzare la configurazione. (chiave CurrentVersionRun) Le vedo con regedit. Con autoruns vedo i programmi allo startup.
- Windows Services: Come i demoni Unix, partono allo startup, creando una configurazione memorizzata in Controlled Cos c'è una Current, e vecchie configurazioni per il ripristino. Creano con privilegi superiori. Se il malware non ha un eseguibile, si appoggia a SVCHOST.EXE, **questa tipologia include anche sagnature**.
- 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 comune) proprio mettendo malware!
- Windows-Hooks: Ad azioni come login, logoff, startup.. viene caricata una dll, che rimane persistenza dopo la prima chiamata.
- Trampoline System Binaries: si usa EXPLORER.EXE (parte a startup, per vedere i file), sempre attivo. Il malware può modificarlo tramite **DETOUR**, ovvero far eseguire inizialmente lista dll ad certo punto, cambiare una parte di esecuzione, e tornare al flusso principale. L'accortezza era nel creare dei riferimenti nel codice, perché la modifica di EXPLORER.EXE può creare problemi. Nel codice detour avremo bisogno di malware-dll che cancella con LoadLibrary, spiazzata di un certo offset da Kernel32.dll, basta crearmi un riferimento. Cosa call prima di fare qualche operazioni per cambiare la finestra malevola, per poi tornare al flusso originale. Attraverso firma programma, **include anche scalabilità privilegi**.
- Dll load order-hijacking: si basa su come EXPLORER.EXE trova le librerie. Cerca in punti precisi, ovvero: cartella applicazione, cartella corrente (dove lancio), cartelle di sistema, path. Se la mia dll malevola si chiama come una ufficiale e la precede in questa ricerca, verrà usata lei, non si sono controlli. Esiste una lista di **KnownDLL** che non funzionano così, se ce ne trovo una dll esterna a tale lista, ma presente in explorer.exe, ho rischio.

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 (diritto di fare determinate azioni). Ogni processo ha un token associato, che viene usato per confrontare con API. Qui ciò che si può fare è usare tool automatici per cercare vulnerabilità, come Metasploit o wineng.

PACKER PIU' RECENTI

- In versioni recenti del packer la chiave non è inclusa nel packer stesso, bensì potrebbe usare forza brutta (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 (sì ok controllo pochi malware), e il malware non deve capire di stari dentro.

COME RICONOSCERE FILE PACKED

Ciò che possiamo fare è:
- Vedere i nomi delle sezioni: con il programma UPX abbiamo sezioni UPX0 UPX1 UPX2 (non standard).
- API scarse: Tipicamente VirtualProtect (dlrit) (regola memoria), LoadLibrary e GetProcAddress.
- Differenze Virtual Size e Raw Size sia molto diverse, è sospetto.
- DAT: se Ghidra mostra molti DAT, vuol dire che non comprende le istruzioni.
- Stringhe poco analizzate, ma se non so quale sono quelle originali..
- Entropia: E' massima se il byte precedente non mi dà alcune info su quello successivo. Normalmente alta l'entropia (valore circa 7 senza packer su E3 toglie cose ridondanti).

SEGRETEZZA

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

- Rootkits user-mode: dirottano Import Address Table (modulo). Un eseguibile può usare API Messagebox. Iscrivo parte di una dll azione binaria direttamente. Nell'eseguibile c'è A7 call cui si esegue jump nella dll a questa API. Il malware può solo bloccare questo salto a questa API, sabendo sul suo codice nascondendo. **Non è un deteur, tocca la IAT, non zona text**. Cambio la firma ufficiale. Sostanzialmente UI solo a ciò che dico io, e poi lo salto a dll originale. (foto 1)
- Inline Hooks: Faccio deteur modificando le dll. Quando ritorno sul flusso originale, cancello le prove. Posso fare deteur in memoria. (foto 2)

EVASIONE

Nel **sistema** si possono essere delle misure che cercano di bloccare le attività sospette. Evado dal sistema. Come?

- Dos-resistant: crollino che cerca la dll malevola. Devo, per iniziare, prendere il PID del processo ultima esistente ed ottenerne l'handle (con OpenProcess) per controllarlo sistematicamente. Il mio processo può allocare memoria al processo ultimo (con VirtualAllocEx per scrivere codice) e poi scriverci (con WriteProcessMemory), o cambiare il suo flusso con CreateRemoteThread. Opero in memoria, non mi scopriono. Attacco sempre esplorer.exe che ha tanti ruoli per le dll.
- Direct Injection: come prima, ma senza dll: solo detour, il codice viene eseguito nel processo. Fattibile se ho problemi di privilegi (processo meno importante modifico pagine di un processo più importante).
- Process Reimplantation: attacco SVCHOST.EXE (è un processo di sistema che può ospitare uno o più servizi del sistema operativo). Windows lo tiene vivo permette di lanciare questo 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 (beato solo i privilegi), usando la ZwUnmapViewOfSection. Sto lavorando in un grande vuoto.
- Hook Injection: Intercettiamo le deviazioni (hook) dei messaggi del processo. Utente crea evento che richiama il SO, ciò comporta un messaggio che viene inviato al processo. Il processo ha una procedura che ascolta i messaggi: come passato al thread dell'applicazione che carica tale dll. CheckForUnicode - LocalFree. Interna al processo, devo fare le cose.
  - Remote Hook low-level: L'app che installa Thook deve rimanere in esecuzione.
  - Remote Hook high-level: Non richiede che l'applicazione chiudano, ma devo rilasciare l'evento. Usato co Keylogger. Ne è esempio: SetWindowsHookEx(WH_CBT,lpfnCBTHook, funzionaleDaLoggare, HandleCorrentProcess, ChiMiLancerhook).
- Kernelmode Hooking Code: Facciamo eseguire nuovo quando un processo è alla, tramite WaitForSingleObjectEX (se presente (dormi ai semafori) il malware passa cose che immobile fare al processo in questione, che non avrà problemi a farlo.

PACKER

INTRODUZIONE

Nascono 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 fa cose diverse dal programma originale, e tramite operazioni di scrambling lo stub viene complicato per ostacolare l'analisi.

COSA FA LO STUB

- Elaborare 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 proteggo.
  - Si mantiene la IAT, ma per ogni DLL si lascia una sola API, poiché l'ordine è prevedibile. Lo stub sa quali sono le altre API da reintrodurre. Quando c'è la richiamo a DLL, ci saltiamo dentro.
  - Non usiamo il SO, quindi mancherà 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 scrive nel wrapper, che verrà chiamato 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.
Possiamo l'immagine creata dallo STUB, che si un file non eseguibile dietro DUMP, in quanto non ha IAT. Stub opera in memoria, sistema gli indirizzi del wrapper, non consegue la IAT. Importiamo la IAT alla fine.

1. Cerchiamo LOAD JUMP. Stub piccolo, lavora su data e in pagine di memoria allocate dinamicamente dal Freep. Qui si aspettiamo un salto lontano all'OEP nell'heap. Usiamo disassemblar: (disassembl Tra trovamo PUSHAD, vuol dire che il packer non sa cosa farà il programma che ora impacchettando, e quindi salva tutto allo stack. Quando avremo settato al programma eseguito, ci sarà un POPAD che ripristina tutto. Ad esempio, ci sarà un JMP EAX, che ci porta all'OEP.
2. Dump della memoria: Ora, siccome fare il dump della memoria del processo, che ora contiene anche il programma originale così riusciamo un file che dovrebbe contenere tutto ciò che serve. 3. Sistemiamo IAT: Con ImportREC (potentissimo), ma richiede OEP e processo attivo. Se corretto, con gentilezza possiamo vedere se la ricrea correttamente.

TROVARE OEP

Ci sono tecniche per trovare l'OEP:
- Tracciamo: Tracciamo in memoria le operazioni, INTO! Ovviamente, Usiamo disassemblar: (disassembl Tra trovamo PUSHAD, vuol dire che il packer non sa cosa farà il programma che ora impacchettando, e quindi salva tutto allo stack. Quando avremo settato al programma eseguito, ci sarà un POPAD che ripristina tutto. Ad esempio, ci sarà un JMP EAX, che ci porta all'OEP.
- Dump della memoria: Ora, siccome fare il dump della memoria del processo, che ora contiene anche il programma originale così riusciamo un file che dovrebbe contenere tutto ciò che serve. 3. Sistemiamo IAT: Con ImportREC (potentissimo), ma richiede OEP e processo attivo. Se corretto, con gentilezza possiamo vedere se la ricrea correttamente.

Malware vs Debugger

Detection

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

- **IsDebuggerPresent**: API usata per vedere la presenza del debugger. Inserisce l'esecuzione del processo che esegue l'API. Possiamo patcharla con disassemblar.
- **CheckRemoteDebuggerPresent**: Simile a prima, ma qui passo passare handle di altri processi.
- **NTQueryInformationProcess**: Usa ProcessDebugPort che ritorna 0 se non sono in debug.
  NTQueryInformationProcess è cariata a runtime, con LoadLibrary per poi prendere l'indirizzo della funzione con getProcAddr.
- **OutputDebugString()**: Se non c'è debug, ritorna un errore. Se c'è debug, ritorna correttamente. Patchato con setEnvVar e poco correttamente. Se errore vecchio e nuovo sono uguali, c'è debugger.

Identifying

Il malware non si rende pienamente conto di essere sotto debugger, ma note del pattern tipici. E indiretto.

- **Ricerca stringhe/variabili**: Controlliamo traccie lasciate dal debugger, come CommandversionDebugging, configurata se c'è debugger.
- **Analisi finestre aperte**: Usiamo FindWindowsA per tornare handler con nome delle finestre, magari il nome della finestra è associata al debugger.
- **Scan della memoria**: Cerchiamo in memoria le operazioni, INTO! Ovviamente, ad esempio ofcharlindi "INTRO" per trovare un byte nell'area di memoria.
- **Timing Checks**: Tramite orologio, vediamo il tempo di esecuzione, se troppo alto, qualcuno ha fermato l'esecuzione per ispezionare.

Interference

Non vado prima a vedere se c'è debugger, queste interferenze ci sono a priori (indipendentemente dal debugger). Ad esempio, ci sono operazioni che non si possono fare in presenza di debugger. Ad esempio, un programmatore lo mette in mezzo a metà point, ma prima c'è inizializzazione del TLS, su cui posso lavorare.
Il malware può anche generare eccezioni singole (eliminabili) o meno singole (INTO). Ciò/DBG capisce quale che ha messo lui.

EXCEPTION VS INTERRUPT

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

Exploiting Vulnerabilities

Il debugger non è esente da bug

- **RVA**: Possiamo avere size 16 del sezione, ma lo modifichiamo con valore più grande. Windows non ha problemi (alloca 16 non ok), CRUDDS si a sovrascrive aree di codice.
- **Raw vs Virtual size**: Non sempre coincidono, CRUDDS oggi prende il minimo ma la dimensione del file su disco o in memoria. Prendo, ad esempio, il minimo tra Raw Size e Virtual Size, avevo problemi.
- **Strings mal processate**: Bug legato a OutputDebugString(msg), se msg = %d%s%x creeha parole CRUDDS crede siano parametri validi, ma ed un certo punto la stringa termina, ci sarà un puntatore a 0 che causa segfault.

Stealth

Il debugger non è esente da bug

- **Breakpoint software**: Usiamo INT 3 (0xCC) per fermare l'esecuzione. Il debugger sostituisce l'istruzione con INT 3, ma se il programma legge la memoria, vede 0xCC. Ciò/DBG ha una copia originale, e ci mostra quella. Se non trovo handler adatto, si usa l'default. La fase è per gli ignoranti, in quanto ogni thread ha il suo stack.
- **Breakpoint hardware**: Sfruttiamo su registri che generano eccezione fra. 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.

Divisione per 0

Sappiamo che FSZO(dove puntare affinandor ysis sistemi, che chiamiamo ID, dia nel introdotto sullo stack con una push. Questo nuovo handler ha campo prev che punta al vecchio handler. Quindi ove FSZO punta sullo stack (dove abbiamo messo il nuovo handler). Questo nuovo gestisce il perd programma per non ritornare mai il controllo.
E' presente anche un handler di sistema che punta ad 0.
Ora, se ho a fare handler di sistema, Se deverbersiamo. due volte EAX, andiamo prima su ID, poi sull'handler più vecchio.

Cosa ho ottenuto?

Facendo questi tipi di trucchi, non riortiamo più sul flusso di esecuzione principale, l'handler non ritorna mai e quindi il codice non continua l'esecuzione. E gestore degli handler torneranno 0 o 1 per come ha gestito l'eccezione, ogi 0 codice stesso gestisce l'handler, quindi non ritorna 0 o 1.

STRUCTURED EXCEPTION HANDLING SEH

Cos'è?

Estensione del linguaggio, offerta da Windows (usato per correzioni low level di tale SO (driver, o caratteristiche di Windows). Nasce da Borland).

Come è fatto?

Mantenimo sullo stack una lista dinamica di gestori annidati, con un campo prev (puntra al precedente) e l'address del gestore.
Sullo stack abbiamo anche thread in questo momento. Se non trovo handler adatto, si usa l'default. La fase è per gli ignoranti, in quanto ogni thread ha il suo stack.

Dove si trova la testa?

Nel Thread Environment Block, ad offset 0 abbiamo pointer al primo elemento. Ci arrivo con FS[0].

Come cresce un nuovo Handler?

Pop EAX ; Ottengo il nome di codeC
Push FS:[0] ; Faccio puntare FS:[0] ad handler1
MOV FS:[0],ESP ; Faccio puntare FS:[0] ad handler1

Malware vs Disassembler

Problema con i jump

- JMP: incondizionato, il flusso passa ad etichetta specifica.
- JE: se ultima operazione ha ritornato 0, allora salta.
- ISZ: se ultima operazione NON ha ritornato 0, allora salta.

Problema

Ho un jz 0, e quindi si crea biforcazione: posso andare all'etichetta ID (se salto), oppure a qa (se non salto, subito sotto). Ghidra procede a disassemblare sempre q. ID lo metto sullo stack e ne prescinde dopo.

Se in q ci fosse jz ID?

Come prima, mettiamo jmp ID sullo stack, e proseguiamo con c, ovvero l'istruzione subito sotto.
Il problema è che, q ID e jz ID = jmp ID, quindi non dovrei mai disassemblare quello che sta ad c; però Ghidra non lo capisce (vede due salti incondizionati) e processa male le istruzioni!

Questo vuol dire che potremmo saltare nel mezzo di un'istruzione, facendole perdere di significato. Ad esempio, ella ed 0-E4 (XS), se saltassi a c1 leggerei "xlf" come "call", e il resto come indirizzo. Ma questa call, con interpreta, non lo eseguirmo mai.

Per risolvere, bisogna disattivare il codice macchina.

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
fra è un esempio 0B FF FC 45, dove:
- 0B FF FC 45 = OR EDI,[EBP+0xFC] ; aggiornamento all'instruction pointer (nuovo indirizzo + contenuto ritornato, nuovo CS) quindi da CS tornamo ad FF.
- FF CC è "INC EAX", perciò Ghidra non accetta che FF faccia parte di due istruzioni. Ciò disattiva il codice.
- 45 è "DEC EAX".

Funzioni dipendenti dai parametri

Tipico dei programmi ad oggetti, i metodi virtuali dipendono dall'istanza (dipendono da parametri). Ad esempio, Ghidra non sa quale sia il metodo virtuale (dipeder dipendente da operazioni dummy) trovamo call ducciato.

Errata traduzione variabili locali

Aggiungendo valori alla stack pointer, si prendono valori di funzioni chiamanti, invece una errata traduzione delle variabili locali. Lo realizzo forzando il disassembler a prendere scelte errate (cmp sempre falso so dico che stack è nella prima pagina) e facendogli compiere operazioni sullo stack.

Fixture fuoco esecuzione

```
call 18
push ebp(..), 5
mov1
18+5: ...
```

Le particolarità è che l'indirizzo di ritorno della call, è 10 stesso. add lunga 4 byte, sommiamo 5 cosi tornamo su ret.
Ghidra non capisce che ritornamo ad eseguire l'istruzione dopo ret.
Per Ghidra la funzione finisce con return, ma noi siamo atterrando lo visione d'insieme, nascondendogli altre call.

Tipi di disassemblatori

Linear:
Non comprende il codice, se associa 2 byte ad una istruzione, non considera eventuali salti nel mezzo.

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

Ottenimento del codice macchina

Codice sorgente
singola unità di compilazione, posso avverra +1

Precompilazione
sostituisco le #include e #define, non risolve.

Compilatore
Ci fornisce codice oggetto. Non ha ancora tutto per eseguire.

Assemblatore
Traduzione in linguaggio macchina parziale, non ancora eseguibile.

Linker
Combina più file oggetto e le librerie di sistema, risolvendo i riferimenti e generando l'eseguibile.

Fila oggetti vs Eseguibile
Associazione tra SEGMENTI (segmenti) e PROCESSI (parti del programma).

Libreria statiche
Scollocate, a compile time, i richiami della funzione vengono sostituiti con codice interno al file eseguibile.

Libreria dinamiche
Codice non inclusa nel eseguibile, ma caricata dinamica a runtime. Linking col SO.

Tipologie di analisi

ANALISI STATICA (malware morto - white box)

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

ANALISI DINAMICA (malware vivo)

- **BASE**: monitoraggio, ad esempio wireShark (black box)
- **AVANZATA**: uso debugger, come OllyDBG. (grey box)

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

Tools utili ANALISI DINAMICA DI BASE

PROCESS EXPLORER:

- **Problema**: un malware può lanciarsi come eseguibile ufficiale, anonimato e assfusca.
- **Cosa fa**: Vediamo chi lancia cosa, con PID, size, il malware si nascondera tra i processi di sistema, lanciati da services.exe. Processo Explorer vede la firma rispetto all'originale, o le differenze tra stringhe.

Process Monitor

Monitora il sistema durante l'esecuzione. Registra le API a chiasi.

Regshot

Esegue due fotografie, pre e post malware.

Wireshark

Osserva il flusso dei dati a livello rete.

ApiMONS

Attaco 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 cosa fa l'applicazione.

Come analizzare

- **1 - Analisi di base statica**
- **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.
- Facciamo a Ghidra un'analisi base, magari per cercare stringhe.
- Regshot per a posteriori modifiche.

NB: Ricerca del main
Se malware è grafico, cerchiamo il ciclo dei messaggi, altrimenti una funzione con parametri compatibili con argc, argv.

Caratteristiche del codice

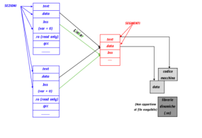
File eseguibili 32 bit Windows

- Formato: PE, portable Executable
- Testato: IMAGE_DOS_mutilia
- Altra testata: IMAGE_NT_WIN_32, espandibile con Ghidra.
- Optional Header fondamentale cpgl. Contiene info come "magic numbers" e RVA di ExportDirectory.

Tipi di indirizzi

- L'RVA (Relative Virtual Address) è l'indirizzo virtuale relativo, cioè lo sfaranno rispetto all'indirizzo di caricamento. Ad esempio, se il mio indirizzo (grafetto) dell'inizio dell'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 assegna nulla.
- **Indirizzo virtuale/logico**: segmento, offset*
- **Indirizzo lineare**: 32 offset*, quindi ho l'indirizzo completo. Modelli flat.
- Pone i segmenti a 0 vuol dire che posso avere sovrapposizione, risolvo con paginazione.



Stack vs Heap

- **Stack**: cresce dall'alto verso l'alto, indirizzi decrescono.
- **Heap**: cresce dall'alto verso il basso, indirizzi crescono.

La memoria stack è una forma di allocazione di memoria che il sistema operativo gestisce continuamente e utilizza per memorizzare variabili locali in ordine LIFO (Last-In, First-Out). Dall'altra parte, la memoria heap è un tipo di allocazione dinamica della memoria che consente ai programmi di strutture dati che richiedono una durata più lunga rispetto alla memoria stack.

Operazioni

- **Push** per lo call, prelevo da instruction pointer.
- **Pop**: per lo return, metto in instruction pointer.

Ordinamento stack

Nozione EBP, posizionato sopra la return, mi serve come riferimento.

```
get EFLAG ecx, ccr ecx, ccr ecr
ccr ecx
ccr ecx
ccr ecx
```