Malware Analysis

November 4, 2021

Contents

Chapter 1

Lezione 1

1.1 Introduzione alla RCE

Reverse Code Engineering del malware, quando si analizza il malware bisogna ricostruire quello che fa a partire dal codice macchina: programma eseguibile o anche un frammento di codice macchina. La pratica di risalire da un frammento di codice al codice sorgente è il RCE.

1.1.1 Definizione di Reverse Engineering

Per definizione, per RE si definisce il processo di estrazione della conoscenza e degli schemi di progetto di qualunque oggetto costruito dall'uomo. È una attività già formalizzata nell'800, quinid ben definita già da molto tempo.
Un modo per capire come funziona l'attività del RE è di accostarla alla ricerca scientifica:

- il ricercatore scientifico accumula dei dati quantitativi, che non sono sufficienti. Bisogna estrarvi una conoscenza più astratta della realtà che c'è dietro e derivare un modello formale che rappresenti in modo formale la realtà che c'è dietro. Dall'insieme e dalle osservabili non è immediato derivare il modello formale, il processo è difficile e richiede intuizione, creatività: non basta leggere i dati, il modello va inventato. La stessa cosa vale per chi fa RE: il lavoro non è di tipo meccanico.

- si usano opportune metodologie, altrimenti non ci sarebbe certezza che la teoria che si sta costruendo è sensata e può ragionevolmente spiegare i dati che si stanno osservando

I due grandi pilastri sono quindi intuizione e metodo.
La più grande differenza è che la ricerca investiga fenomeni naturali, mentre chi fa RE si concentra su gli artefatti umani: il ricercatore scientifico può fare delle ipotesi che non vengano semplicemente spiegate dal cervello, che ha dei limiti dal punto di vista dei modelli che può elaborare. Il ricercatore scientifico non ha certezza di trovare una teoria ed una forma a cosa sta osservando, mentre per chi fa RE c'è questo vantaggio. Quindi, concretamente c'è la garanzia che questa conoscenza era posseduta da qualcuno e che può essere riscoperta.

1.1.2 Reverse Code Engineering

È il processo di ricostruzione delle finalità, degli algoritmi, delle strutture dati implementate da un programma per un calcolatore elettronico.

Ricordiamo che, con calcolatore elettronico si intende un qualunque dispositivo programmabile, non per forza un PC o laptop. Ci sono vari modi di riferirsi al RCE, come de-compilazione, ingegneria inversa dei programmi, in generale si parla di reversing.
Vedendo il reversing come una black box, l'input del RCE è una rappresentazione "a basso livello" del programma per il calcolatore, come ad esempio il file eseguibile contenente il codice macchina. L'output può essere variegato: è una rappresentazione a più alto livello, ma non possiamo dire con esattezza cosa debba esser in quanto quando si fa reversing non c'è un singolo use case:

- conoscere gli effetti del programma, allora l'output è la descrizione degli effetti del programma
- voglio esaminare il protocollo di comunicazione, otteniamo la descrizione del protocollo
- etc...

Si passa sempre da un livello di bassa comprensione ad uno più altro.
Ma alto e basso sono relativi, quindi occorre capire e definire cosa si intende per alto e basso livello.

Motivazioni del reversing

Cerchiamo di capire perché si fa reversing, non tutte le motivazioni hanno lo stesso livello etico o valore legale. Alcuni casi di utilizzo:

- abbiamo scritto un programma perdendo i sorgenti. Facciamo il reversing del programma per capire cosa faceva, questo è lecito. Se il programma è dell'azienda ed ho il consenso è lecito. È una attività che spesso si usa per programmi legacy in azienda.
- programma di terzi, commerciale, che deve interagire con un altro programma scritto da noi. Vogliamo fare in modo che i due programmi si interfaccino bene, può essere necessario fare reversing sul programma per vedere come integrarlo meglio. Se è lecito o no dipende da licenze e legislazioni del paese
- programma scritti da terzi, facciamo reversing per capire come rendere un programma simile più efficiente e compatto. È ancora meno lecito del caso precedente, tipicamente l'altra parte può dire che il lavoro viene sfruttato per avere vantaggi
- siamo in una azienda che tiene alla sicurezza dei dati, per l'azienda è un problema affidarsi a programmi di 3° parti. Come fa l'azienda a fidarsi che non vengano esfiltrati i dati dai dischi da parte del programma utilizzato? Uno dei modi per verificare che sia tutto apposto è fare il reversing dell'applicazione. Sembra lecito, ma tutte le licenze dei programmi commerciali la vietano, quindi a rigore non si può fare
- superare i meccanismi di protezione digitale, se c'è un sistema di protezione e si fa il reversing si sta commettendo un atto illegale.

Si fa reversing per analizzare il comportamento del malware per analizzarne gli effetti, o per produrre delle forme di protezione verso di essi come gli antivirus: gli antivirus sfruttano due grandi tecniche per il funzionamento

1. riconoscere la firma del virus

2. riconoscere una sequenza di operazioni "tipiche" fatte dai virus

il problema è fare il reversing dei virus in circolazione per poter costruire l'antivirus.
Si può fare reversing del malware anche per scrivere nuovi malware, in modo da cambiare le firme ed esulare gli anti-virus appena diffuso
Un'altra attività di reversing viene fatta sui SO, al fine di attaccarli ad esempio tramite 0-day (ovvero attaccare con dei malware mai visti prima), o anche per rendere il SO più robusto.
C'è poi un'area del reversing che si occupa delle debolezze degli algoritmi crittografici, nella loro implementazione.
Alcune di queste attività sono illegali, alcune addirittura in quasi tutte le nazioni del mondo. Ci sono attività legali o tollerate, perché il punto chiave è il perché si fa questa attività ad esempio a scopri di didattica, anche se fatto su programmi commerciali. Anche se nelle licenze d'uso c'è scritto che non è possibile fare RCE, questo può essere vessatorio, quindi poi la decisione finale spetta al sistema giuridico. **esempio:** sembra essere consentito in Italia fare reversing sul software di una stampante al fine di vendere cartucce di inchiostro riciclate. Le stampanti bruciano dei bit del componente elettronico della cartuccia quando questa è terminata, quindi occorre capire come questo viene fatto dal software.
La cosa diviene illegale se io faccio la stessa cosa per vendere cartucce nuove compatibili con la stampante.
L'attività ed i tool sono sempre gli stessi, ma quello che cambia è **il perché** viene fatta: spesso il confine fra legalità ed illegalità è sottile. L'attività di RCE non è lecita o illecita di per se, è lo scopo che se ne fa che la rende tale (Ricorda la differenza fatta a SERT fra hacker e cracker).

Principi generali del RCE

IL RCE può essere considerato l'operazione inversa della programmazione: nella programmazione l'idea di algoritmo viene tradotto in linguaggio di alto livello e poi compilato in codice macchina; in RCE si fa l'opposto.
Un programmatore segue dei principi nello scrivere il codice, quanto più si adottano correttamente i principi, quanto più si fa bene. Anche nel RCE vanno seguiti una serie di principi e regole generali per fare un buon lavoro. I principi generali:

Maggiore è la comprensione dell'interno sistema, tanto più rapida ed efficiente è l'attività di reversing. Non si può fare reversing di qualcosa di qui non si sa nulla, l'hacker che fa il reversing deve essere competente in diversi settori dell'informatica: SO, architetture dei calcolatori, etc... Meno cose si sanno, più difficoltà si avrà nel lavoro di reversing. Altre competenze da avere è capire il processo di compilazione del programma, come i costrutti di alto livello vengono tradotti in assembly, siccome i dettagli dipendono dal compilatore cambiano a seconda del compilatore. Il formato del file eseguibile è un altro elemento importante: una cosa sono le istruzioni macchina, un altro è il contesto di esecuzione del file eseguibile, ovvero di come poi il programma verrà effettivamente eseguito. Inoltre, il lavoro di reversing è una "battaglia di teste" e siccome chi ha scritto il malware non ha interesse che sia analizzato, mette in piedi una serie di misure di offuscamento, protezione, rilevazione del debugger, in macchine virtuali e sandbox e altro. Chi fa il malware vuole proteggersi da chi vuole analizzarlo, inventando sempre nuove soluzioni ed il lavoro è cercare di capire e superare queste soluzioni.
  Per questo motivo, fare reversing è sempre un atto creativo, dove si usa sempre la testa

ma va comunque supportato dalla conoscenza e quindi richiede un continuo sforzo ed aggiornamento.

Per capire il codice scritto da alti, è necessario capire come funziona il proprio. Se chi ha scritto il malware ha necessario, ad un certo punto, una struttura di dati dinamica, come un albero bilanciato, è necessario sapere come gestire e implementare una struttura dati dinamica. L'obiettivo del processo di reversing non è capire cosa fa una singola istruzione macchina, bensì tutto ciò che fa il programma per capire cosa pensava chi l'ha concepito. È essenziale saper programmare bene

L'attualità e la conoscenza dei tool di reversing determina la qualità del processo di reversing. Le applicazioni moderne sono costituite da una grande quantità di codice macchina, che non è gestibile a mano. Ormai sono sempre più grandi di ciò che si può gestire senza strumenti sofisticati, quindi bisogna saper usare gli strumenti giusti. Oggi, ghidra è uno dei principali strumenti per fare RCE (open source). Gli strumenti sono sofisticati, la curva di apprendimento non è del tutto lineare, per cui è necessaria molta pratica per saperli usare.

La chiave del reversing è la capacità di identificare e comprendere gli schemi ricorrenti nel codice, di conseguenza non esiste sostituto dell'esperienza. La bravura di chi fa RCE si misura nel numero di ore dedicate al fare reversing. È sempre un principio generale, tanto più si ha esperienza, quanto più è rapido fare RCE.
I compilatori producono i costrutti di alto livello in certi pattern di codice macchina, con i decompiler si ottengono i pattern ad alto livello ma non è detto che il codice ad alto livello sia più facile da capire cosa il programma fa dal codice ad alto livello. L'abilità principale dell'hacker esperto è quella di saper riconoscere le strutture nascoste nel linguaggio macchina

I principi generali di per se non sono sufficienti, serve anche metodologia. Alle volte si dice che la programmazione è arte, compresa solo da altri programmatori. In effetti, anche l'attività di RCE può essere considerata una forma d'arte perché richiede intuizione, capacità di problem solving. Punto importante: metodo, costanza, tempo e impegno mettono in condizione chiunque di poter fare questo mestiere.

Metodologia

Non si può prescindere dalla metodologia, bisogna avere ben chiari gli obiettivi. Bisogna avere chiaro la domanda a cui dare risposta e deve essere chiara, perché se ci cerca di scoprire tutto, può richiedere moltissimo tempo. Occorre stabilire in che modo ottenere l'obiettivo: se c'è un virus pericolo che può infettare la macchina, non c'è cosa più pericolosa di essere infettati sulla macchina con cui si fa reversing: non deve mai avvenire, quindi non è detto che si può operare sul malware eseguendolo. Inoltre, essendo un processo creativo, bisogna di continuo verificare la correttezza di cosa si sta facendo. Tutto ciò rende necessario formalizzare una metodologia, meglio se scritta. Gli approcci fondamentali del reversing:

- analisi "black box" o live code analysis: eseguo il programma e cerco di capire cosa fa, in ambienti più o meno controllati. Non è sempre possibile farlo e non può fornire informazioni su porzioni di codice non eseguite

- analisi white box: analizzo il codice e cerco di capire cose fa "guardando nella scatola". Il problema dell'analisi è il costo in termini di effort e di tempo
- analisi mista o gray box: approccio che in linea di principio combina metodi white box e black box, mischiando i due livelli. Il principale tool che si usa in questa fase è il debugger, in generale la metodologia è molto efficace e quindi nei malware ci sono una serie di elementi per cercare di renderla difficile.

Esempio di metodologia generale in 9 passi:

1. descrizione preliminare: descrivere cosa si sa, da dove viene il malware, cosa ha prodotto etc... Tutto quello che si sa va scritto
2. formalizzazione dell'obiettivo: non possiamo pensare di sapere tutto. Decidere cosa scoprire, ad esempio l'IP a cui si collegava, che file ha esfiltrato. Passo cruciale, perché l'attività di reversing va centrata su questo passo, inoltre permette di definire quanto tempo ci metterò a fare reversing
3. ottenimento del codice macchina: può essere immediato, alle volte il codice è offuscato o protetto con cifratura. È un passo non banale, alle volte è necessario fare il de-offuscamento a mano
4. osservazione del funzionamento: se posso, faccio analisi black box, è analisi dinamica. In alcuni casi non posso farla, se il codice ad esempio sfugge al controllo o se non posso eseguirlo
5. disassemblaggio white box, se non ho risolto il problema al passo 4, tipicamente si usano disassemblatori interattivi, ovvero che consentono di interagirvi per indirizzarlo nel suo lavoro. Può non funzionare correttamente, perché sono state usate delle contro misure
6. localizzazione del frammento assembly: trovo il frammento che può rispondere alla domanda. Servono delle tecniche per trovare il punto di interesse. Sono varie le tecniche per fare il passo
7. analisi del frammento assembly: una volta trovato il punto, si cerca di comprenderlo. È la fase più critica, è necessario ed essenziale annotare tutto ciò che si trova e se durante la fase ci sono punti che tornano ad essere interessanti si reiterano i precedenti punti
8. verifica dei risultati: occorre verificare che quanto scoperto è corretto
9. riepilogo in un report: viene riepilogato tutto ciò che è stato fatto, cosa si è appreso, cosa si è ottenuto etc...

1.2 Concetti fondamentali di analisi del malware

Per fare analisi del malware, ci sono due grandi possibilità per quanto riguarda i sistemi host:

- Linux
- Windows

La nostra attività di reversing è focalizzata sull'analisi del malware e questo attacca nella maggior parte dei casi sistemi Windows, ma la scelta di usare Windows come sistema per analizzare il malware non è per forza la migliore: su Linux non è possibile eseguire il malware per natura, quindi si evitano eventuali pericoli di esecuzione del malware sulla macchina host.
Non vuol dire che non esistono malware per Linux, ma sono rari ed a quel punto è possibile analizzarli spostandosi in ambienti Windows. D'altra parte, è comodo avere anche degli ambienti che esegua in maniera controllata il malware: se il sw è Windows-based, serve anche un ambiente Windows. Si può quindi usare come base un host Linux ed avere Windows come VM:

- è possibile scegliere la versione di Windows più adatta da installare sulla VM, magari per far girare dei malware patchati da determinate versioni di Win (ne usiamo quindi una vulnerabile)
- l'host viene usato per l'analisi, mentre la VM per l'esecuzione

Possono esserci diverse scelte come VM per il reversing:

- VmWare, molto usata per il reversing
- VirtualBox
- QEmu, rispetto alle prime due, questa è disponibile solo per Linux

Il vantaggio della VM è che è possibile costruire degli snapshot: se viene perso il controllo e la macchina è infettata, è possibile ripristinare lo snapshot per riparare al danno.
È possibile (teoricamente) che il malware si diffonda da una VM Windows al sistema host sempre Windows: tipicamente un malware ha delle contromisure per capire se sono in esecuzione sulla VM o sul sistema host. È necessario da parte dell'analista capire come fa il malware a vedere che è in esecuzione su una VM ed inibirlo. Il malware può fare un numero limitato di controlli, il grande vantaggio di chi fa l'analisi è che avendo il controllo sul codice può patcharlo e farlo eseguire sotto specifiche condizioni.
Ci sono comuqnue dei casi in cui è possibile che si riesca ad infettare il sistema host dalla VM, ma in ogni caso l'esecuzione in VM è un grande passo avanti di sicurezza. Inoltre, può accadere che l'antivirus presente nella versione di Windows abbia un anti-virus per-installato che magari blocca il malware, quindi andrebbe disabilitato per un tempo limitato per evitare che ciò accada.

1.2.1 Come fare RCE

Possiamo vedere il RCE come una scatola nera, in cui entra codice macchina ed esce una rappresentazione di alto livello che vogliamo ottenere dall'analisi. Il problema sta proprio nel codice macchina, per poter fare reversing occorre capire come è fatto e da dove viene fuori.
Il codice macchina è generato nel processo di compilazione:

- la compilazione parte dal file sorgente
- i primi componenti che processano il file sono il **pre-compilatore**. Ad esempio, in C esiste la direttiva `include`, o definire le macro, il pre-compilatore si occupa di aggiungere gli include e sostituire le macro nel file. Il risulato passa ad un secondo parser chiamato **compilatore**

- l'output della comilazione è il file oggetto, che non è un programma eseguibile, che contiene le istruzioni macchina corrispondenti al programma nel file sorgente.
- Questo meccanismo viene ripetuto per ognuno dei file sorgente, tutti i file oggetto finiscono al **linker**, il risultato è un eseguibile.

La differenza fondamentale fra file oggetto ed eseguibili sono i riferimenti: nei file oggetto, tutti i riferimenti ai file sorgente sono appesi, ad esempio le chiamate alle funzioni di libreiria. Non si possono risolvere, il linker mette assieme il contenuto dei file oggetto. Ad esempio, mettere a posto i riferimenti tra diversi file sorgente o le librerie.
Nelle librerie ci sono una serie di funzioni già pronte che vanno collegate al file sorgente (es: printf, scanf etc...) ed è il linker a predisporre il collegamento.
Il file eseguibile non contiene solo le istruzioni dei file sorgente e delle librerie, ma anche concetti relativamente al processo (vedi meglio); inoltre c'è differenza fra librerie statiche e dinamiche:

- nelle librerie statiche, ogni volta che serve una funzione, si cerca nell'opportuno file il codice che realizza la funzione e viene copiato nel file eseguibile il codice macchina necessario
- nel collegamento dinamico, il linker registra nell eseguibile che una certa funzione verrà fornita a run time dal SO, ad esempio le dll di Windows o shared object di Linux. Quindi, alla "creazione del processo" verrà fornito il codice da eseguire

Nell'analisi del malware, le librerie dinamiche sono un vantaggio da un certo punto di vista ed uno svantaggio da un altro lato: non è banale capire cosa fa del codice, se è linkato staticamente magari è una banalità ma essendo linkato staticamente è difficile capire cosa fa. Se la libreria è dinamica ed il malware fa riferimento alla funzione, capisco dal nome cosa fa la funzione. In generale è un vantaggio se il malware usa librerie dinamiche, anche perché tipicamente l'eseguibile risultante è più piccolo se ci sono librerie dinamiche. È uno svntaggio se chi fa l'analisi del malware non riesce a capire quali sono le librerie dinamiche che il malware usa e questo può diventare un problema.
Quindi, non c'è una risposta esatta: entrambe le opzioni hanno vantaggi e svantaggi.
Abbiamo quindi del codice macchina, a questo punto va analizzato

- l'analisi statica cerca di prendere il codice e capire cosa fa, ci sono due grandi tool per fare questo lavoro
  - disassemblatore, che possono essere
    - lineari, ovvero un tool poco complesso da capire e da utilizzare. Va bene per piccoli malware ma diventa più difficile fare dei lavori più complessi. Esempi: objdump di Linux
      Nell'architettura dei calcolatori (di Von Neumann), i dati ed i programmi sono memorizzati nella stessa memoria, quindi l'eseguibile ha sia dati che istruzioni macchina. Se trovo un byte "103", come lo interpreto? È un dato, una variabile pari a 103? Non c'è modo di capire cosa sia, e questa è la grande differenza fra i disassemblatori lineari e quelli interattivi: il disassemblatore lineare parte dalla prima istruzione macchina che viene eseguita quando si lancia il programma e da lì scende, andando per sezioni. Se nel processo c'è un salto nel codice e in mezzo ci sono stringhe o altro, il disassemblatore va avanti e continua ad analizzare quel codice come se fossero istruzioni macchina.

    * interattivi: ad esempio Ghidra ed IdaPro (di cui esiste una versione gratuita, ma sta bene dove sta). Il disassemblatore interattivo parte dall'entry point del programma e segue il flusso: dove trova un salto, sa che c'è codice, se trova una chiamata a funzione c'è codice. Dovunque non arriva, lascia in incognito. È interattivo perché è l'analista a dire che in determinati punti c'è codice per far si che il disassemblatore conitnui, questo implica un uso corretto del tool, fornendo informazioni all'interno di essi.

Quando viene costruito un file oggetto o un esegubile, viene organizzato in modo da riconsocere le informazioni: un file oggetto ha un certo numero di sezioni:

- testo: c'è il codice macchina
- data: contiene variabili inizializzate con valore diverso da 0
- bss: contiene la descrizione delle variabili inizializzate a 0
- read only: dati costanti, ad esempio le stringhe esplicite

Nel file esegubile, più sezioni sono collassate in segmenti dove ogni segmento è descritto dal modo in cui il SO deve preparare le pagine che contengono i dati e le istruzioni, e potrebbero andare a finire nella stessa pagina perché devono essere pagine read only. In architettura Intel non ha quindi molto senso differenziare testo e read only, poi quando si porta in memoria ci sarà il processo per cui quando si porta in memoria ci saranno le diverse pagine corrispondenti ai segmenti.
Windows e Linux usano formati differenti di eseguibili:

- ELF per Linux
- PE-Coff storicamente, oggi sono PE(32 bit) o PE+(64 bit)

È importante sapere del formato del file esegubile, chi scrive malware sa che chi lo analizza usa un debugger interattivo e quindi stravolge il formato del file eseguibile per evitare che chi lo analizza capisca come è organizzato il file. Quindi, provo ad usare il disassembler e quello non capisce niente (cit): magari parte dall'entry point sbagliato o altre conto misure.

Eseguibili a 32 bit o 64 bit

I processori Intel sono nati a 32 bit, ma uno dei dogmi commerciali è la compatibilità verso i programmi precedenti, quindi c'è la capacità di processori moderni di eseguire programmi a 32 bit.
Quindi, ogni pagina di memoria ha un bit che indica se questa ha istruzioni a 32 o 64 bit ed il processore è in grado di fare swapping fra le due dimensioni e quindi di avere SO che supportano esegubili a 32 e 64 bit.
Dobbiamo quindi gestire due ISA diversi, uno a 32 ed uno a 64 bit. La gran parte del malware è ancora scritta a 32 bit: se scrivo malware è perché lo voglio diffondere il più possibile, in modo da farlo girare sia su nuove che su vecchie macchine. Quindi ancora oggi gran parte del malware è a 32 bit, ma cominciano a diffondersi anche malware a 64 bit.

Little endian vs Big endian

I processori Intel sono little-endian: il byte è l'unità di base del processore, che vengono aggregati per rappresentare valori più grandi:

- word: 4 byte
- quad-word: 8 byte
- etc...

Nell'interpretazione little endian si mette all'indirizzo di memoria più piccolo il valore meno significativo del numero, mentre big endian è l'opposto. **esempio: 515** è fatto di due byte: 2 3 (2*256 + 3). Se il valore è codificato in little endian, viene rappresentato in memoria come 3,2 mente in big endian abbiamo 2,3 e può accadere che nel programma si usi un protocollo di rete e se va usato un valore va codificato in big endian, quindi il disassemblatore non lo sa e deve essere l'analista a scoprirlo.

Numeri esadecimali

Base 16, cifre da 0-9 e lettere da a-f. Importanti perché ogni cifra esadecimale è mezzo byte, quindi è immediato prendere un numero exdec e capire la maschera di bit: 515 = 0x(0)203. Ghidra usa la sintassi assembly della Intel (ufficiale dei manuali), mentre un tool come objdump usa la sintassi AT & T. Il compilatore gcc ha una toolchain interna che usa AT & T, quindi può capitare di dover fare reversing di codice che viene fornito con questa sintassi. La grande differenza fra le due sta nell'ordine dei parametri (src e dest), poi ci sono altre differenze minori.

1.2.2 Attività di reversing

Tentativo di apertura di un programma con Ghidra (fatti un hello world ed aprilo).
Dopo l'analisi del file, ghidra elenca alcune cose iniziali:

- header, dove è inidicato come creare il processo una volta che il programma va in esercizio. Nell'header PE ci sono alcune tabelle, la prima delle quali non è molto interessante poiché descrive il file come se fosse dos (ma ormai i file dos non si usano più)
- IMG_NT_HEADERS32: i primi patrametri della tabella sono una signature, che permette di riconoscere il file come PE.
  Ci sono poi due tabelle principali chiamate IMG_FILE_HEADER ed altra, che contengono infromazioni molto importanti:
  - architettura della macchina
  - timestamp di quando è stato costruito il file
  - address o entry point: prima cosa essenziale da capire. Accanto all'interpretazione, sono mostrati i byte corrispondenti. Ghidra quindi assegna una label ad ogni indirizzo e questa è significativa se ha scoperto a cosa fa riferimento. Doppio click sulla label porta nel punto del file corrispondente.

La "entry" è una posizione nel file, ma anche una posizione in memoria: c'è una doppia visione di ogni cosa di cui si fa reversing

- visione fisica, del file esegubile
- visione della memoria (RAM) associata al processo quando il SO esegue il file eseguibile

Queste due finestre sulla realtà vanno di pari passo: il file eseguibile è diviso in segmenti, avremo quindi dei segmenti per code, data, bss e read data e ciascuno di questi corrisponde a delle pagine di memoria usati per contenerli, avremo quindi delle pagine dedicate al codice che corripsondono alla zona del file dove ci sono le istruzioni. Un'altra parte sarà dedicata ai dati, che corrispondono però sia al segmento dati che al bss poiché le variaibli sono trattate similmente.
Stesso vale probabilmente (ma dipende dall'implementazione) anche i dati readonly finiscono in una zona dati, perché la protezione sulla scrittura dei dati è implementata a compile time, ma non c'è reale necessità ad imepdire che quella zona di memoria non venga scritta perché nel programma non ci sono istruzioni che le scrivono.
Ci sono altre almeno due zone di memoria, ovvero stack ed heap:

- heap: zona per l'allocazione dinamica della memoria, in modo quasi arbitrario
- stack: zona presente in tutti i programmi, di taglia fissa e che serve per due scopi: realizzare il meccanismo di invocazione delle funzioni e correlato a questo, è usato per allcore le variaibli automatiche:
  - variabili globali finiscono nella zona data. Visibili in tutto il programma e persistenti finché il programma gira
  - variabili locali o automatiche finiscono sullo stack, visibili sono nella procedura dove vengono definite e persistenti finché la funzione viene eseguita. Il meccanismo più comodo per allocarle è lo stack

  Bisgona capire come tutto questo viene getito a livello assembler. Ci sono poi altri tipi di variabili

  - statiche e globali: la presistenza è uguale, ma il fatto che sia statica implica che la visibilità è locale all'unità di compilazione, ovvero è visibile sono nel file dove viene dichiarata
  - statiche e automatiche: la visibilità è identica, ma la persistenza cambia, in quanto la variabile conserva il suo valore anche dopo che la funzione smette. Dove viene allocata la variabile: nei data. Lo stack cresce e decresce di continuo e contiene dati temporali. Lo heap è per allocazioni dinamiche, quindi viene messa nel blocco data perché dal punto di vista del funzionamento delle istruzioni macchina non c'è differenza fra variabili statiche e globali, la differenza sta a compile time nella visibilità

Indirizzi di memoria

Abbiamo sempre due visioni: memoria e file eseguibile, cerchiamo però di essere più attenti: in Ghidra, accanto ad ogni costrutto c'è un valore numerico che è l'indirizzo in RAM del costrutto. Per passare dalla posizione sul file all'indirizzo in RAM ci sono diversi passaggi: il linker mette insieme una serie di file oggetto, con le loro sezioni di testo, data etc... e mette insieme tutto il materiale delle sezioni di testo in un unico segmento e comincia a calcolare gli offset delle

istruzioni macchina all'interno del file. Costruisce quindi un file eseguibile che si presuppone venga caricato ad un certo indirizzo.
Supponiamo di avere: l: mov eax, [h], a livello di Assembler lavoriamo ancora con i nomi simbolici. Sarà il linker a trasformare l in un indirizzo, e tutte le istruzioni dopo seguiranno di cosneguenza: conta quindi l'indirizzo dove verrà caricato tutto una volta che li programma va in esecuzione.
Prima, l'indirizzo di caricamento era fissato nel SO, inoltre programma differenti potevano andare in conflitto fra loro. Oggi è tutto risolto tramite meccanismi hardware e software, l'idea fondamentale è che quando il linker alla fine arriva a generare il codice macchina, questo è basato sugli **indirizzi logici**. Il processore Intel ha la memoria divisa in segmenti, ogni segmento contiene del codice e per far riferimento ai segmenti c'è un registro di processore apposito, cs. In Ghidra, noi troviamo l'offset rispetto alla base in memoria di dove è messo il segmento codice. Nel caso sopra, h è un indirizzo di un dato, che starà nel segmento dati, che in RAM è identificato da un altro registro segmento che è ds, quindi anche in questo caso è l'offset rispetto al segmento dati.
Windows, come molti altri SO moderni, non usa la segmentazione a pieno ma adotta un meccanismo flat memory: l'indirizzo di base di tutti i segmenti è 0, quindi l'offet nel codice macchina è formalmente un offet del segmento, ma è anche un identificativo univoco della memoria. (offest + 0 = offset).
Col modello di memoria piatto però, non si può più avere offset uguali, perché tutti partono da 0 e quindi c'è un problema: lo spazio di indirizzo delle istruzioni macchina, dello stack, dei dati etc... coincidono.
Ora, partendo dall'indirizzo logico, tramite l'unità di segmentazione arrivo all'indirizzo lineare che non è ancora quello finale perché c'è un altra unità hardware che è la paginazione: tramite le Page Table, è possibile tradurre l'indirizzo finale in un indirizzo fisico che viene usato nel bus per accedere ad una certa cella della RAM.
Di fatti, nei SO moderni i segmenti principali sono tutti a 0, quindi quello che veramente lavora è l'unità di paginazione: la funzione principale è realizzare un meccanismo di protezione della memoria:

- un programma viene caricato dal compilatore tipicamente ad un indirizzo prestabilito, che è un indirizzo logico
- la paginazione fa si che ogni processo abbia delle PT differenti e che mappino su indizzi fisici differenti ovvero su celle di RAM differenti

Si parla di **indirizzi virtuali** quando il mapping fra indirizzi fisici e indirizzi lineari non è fissato ma cambia di continuo. Nel codice macchina siamo ancora a livello di indirizzo logico, e bisogna tenere presente il meccanismo che lo tradurrà in un indirizzo fisico.
Andando all'indirizzo indicato da "entry", si arriva ad un indirizzo (CHE È IN LITTLE ENDIAN) a cui va sommato l'offset dove viene caricato il file per ottenere l'indirizzo logico.
Non è detto però che quando il programma in Windows, quando lanciato, viene caricato nell'indirizzo prestabilito ma può capitare che il file sia rilocato

Regstri del processore

A 32bit, abbiamo i seguenti registri:

- registri general purpose

  - eax (dove ax è a 16 bit, ah ad 8 bit e al a 4 bit)
  - ebx
  - ecx
  - edx

  tutti con le contro parti a 8, 4 bit

- altri registri
  - esi
  - edi
  - ebp
- registri segmento
  - cs
  - ds
  - es
  - fs
  - gs
  - ss
- usa serie di registri speciali
  - esp: stack pointer
  - eflags: bit che contengono lo stato corrente del processore
  - eip: istruction pointer, non è un registro visibile direttamente, si può manipolare solo con istruzioni di salto e decisioni
- registri per manipolare lo stato interno dei processori
- registri per manipolare i co-processori: un tempo i componenti con registri floating point erano venduti separatamente
  - fpr0 - fpr7 (mmx0 - mmx7)

In realtà tutti i processori moderni hanno una ISA a 64 bit, i registri hanno anche una parte composta da 32 bit in più ad hanno la r davanti al nome. Inoltre, vengono aggiunti anche nuovi registri r8 - r15.

Gestione dello stack

Lo stack dei processori è full descending: il processore lavora a 32 bit ed ogni cella di stack è di 4 byte. Lo stack cresce dal basso verso l'altro, esp in ogni momento contiene l'indirizzo dell'ultimo elemento, ovvero di quello riempito (lo stack è LIFO) da cui si dice che è **full** perché punta all'ultimo elemento inserito. Descending perché gli indirizzi crescono mano a mano che si scende verso gli elementi più vecchi, quindi ogni volta che metto un nuovo elemento lo inserisco in un indirizzo più piccolo di quello più vecchio.

1.2.3 Chiamata a funzione

Le funzioni (sia in archiretture a 32 che a 64 bit) si appoggiano allo stack. Supponiamo di usare una certa parte dello stack e troviamo una CALL ad un certo indirizzo (o etichetta) f, cosa fa il processore: le call possono essere

- nello stesso segmento
- fra segmenti differenti

il risultato pratico è che ci sono più tipi di call. In genere, vediamo quelle nello stesso segmento in quanto lo spazio di indirizzi può essere largo 4G di celle e quindi sufficienti. La CALL quindi salva l'indirizzo a cui ritornare subito dopo aver fatto la chiamata, nello stack viene quindi aggiunto l'indirizzo di ritorno h ed aggiornato esp.
Dopo di che, viene caricato l'indirizzo f in eip, ovvero nell'istruction pointer, quando f termina lo fa con la `ret`, che prende l'indirizzo puntato da esp, lo carica in eip e poi incrementa esp in modo che punti alla locazione che sta sotto (eseguendo quindi la pop). Se la funzione fa cose sbagliate e fa una return quando esp punta ad altro e non ad h succede qualcosa di inatteso. Le funzioni devono tenere lo stack bilanciato, quindi quando si fa la return bisogna tornare ad h, ma "giocare" con lo stack è uno dei tanti modi con cui si tenta di inibire il disassemblatore per non seguire più la logica generale.
Quindi, una cosa è l'idea di programmazione ad alto livello, un altra è quello che riguarda il codice macchina.

Funzioni di diversi tipi Ci sono alcuni tipi di funzioni, come ad esempio quelle di tipo "thunk", che sono di librerie dinamiche (DLL) che ad esempio usano una sola istruzione per fare una jmp ad una locazione che sarà scritta a run-time

Convenzione per la chiamata a funzione

Non c'è un unica convenzione con cui avviene il passaggio dei parametri, a seconda del SO cambiano ma anche in base ad altri fattori. Le due principali sono

- convenzione del linguaggio C, ovvero cdecl
- convezione stdcall, la modalità standard con cui il SO ed i compilatori Microsoft fanno le chiamate a funzione

CDECL: andrebbe distinto fra 32 e 64 bit, a 32 bit i parametri sono passati sullo stack in ordine inverso rispetto all'ordine logico della dichiarazione della funzione (perché così poi la pop ristabilisce l'ordine). Siccome vengono fatte delle push sullo stack, questi sono dati temporanei e quindi poi vanno recuperati: questo, nella convenzione cdecl va fatto dal chiamante, quindi ci saranno le pop, ma verosimilmente si fa un `add esp, 0xc`

STDCALL: identica a CDECL, tranne per il fatto che chi toglie i valori dallo stack è il chiamato. Le istruzioni Intel permetto di fare questo con la `ret 12`, ovvero toglie 12 byte e ritorna al chiamante (dopo aver ovviamente tolto l'indirizzo di ritorno e messo in eip).

Stiamo parlando di un'architettura x86, vengono messi gli argomenti in ordine inverso e poi

viene fatta la call e viene messo l'indirizzo di ritorno sullo stack, ma questo serve anche per scrivere le variabili del chiamato: le variabili automatiche non hanno dei valori iniziali, quindi viene fatta una sub per poter riservare alle variabili lo spazio necessario. A questo punto ESP punta alla locazione dove andrà var2 (una delle due variabili della funzione). Se però la funzione vuole leggere l'argomento 1, bisogna fare i calcoli: starà in ESP + 0xtot, dove tot è dato da cosa è stato inserito nello stack quando viene preso il chiamato. Il compilatore fa tutto automaticamente, per motivi storici i compilatori usano un registro particolare chiamato EBP, che punta al **frame** sullo stack: il frame identifica la porzione dello stack dedicata ad una specifica funzione

Funzionamento di EBP

Quando viene chiamata la funzione, viene salvato EBP nello stack (push), perché immaginiamo che anche la funzione chiamata abbia un registro EBP e quindi all'uscita va rimesso come era prima.
Quindi, la prima cosa che viene fatta dalla funzione è fare un push di EBP, ovvero mette il vecchio valore di EBP, quindi ESP punta sul valore di EBP. Si sposta poi il contenuto di ESP in EBP, ovvero adesso EBP diventa = ESP, ESP puntava su old EBP e quindi EBP punta lì. Adesso è possibile muovere ESP come si vuole, perché EBP rimane fermo in quel punto dello stack, quindi la funzione fa una AND di ESP ed un valore esadecimale 0xfffff0, che in complemento a 2 è un valore negativo e quindi fare un AND logico vuol dire azzerare gli ultimi bit e l'effetto è l'allineamento dello stack ad un multiplo di 16, in quanto non siamo sicuri che lo sia.
La sub alloca spazio nello stack di 10 ex, ovvero di 16 byte e quindi di 4 variabili, EBP rimane come prima, mentre ESP si muove e punta sull'ultima variabile allocata. Ora per identificare gli argomenti della funzione, si usa EBP:

- argomento 1: EBP + 8
- argomento 2: EBP + 12
- argomento 3: EBP + 16

Ma a questo punto ESP si può usare a piacere e modificarlo come si vuole, e si usa EBP per fare l'accesso alle variabili allocate prima, stavolta con EBP - 0xtot (RICORDA CHE LO STACK CRESCE PER INDIRIZZI DECRESCENTI), quindi è possibile accedere alle variabili in modo uniforme.
La sequenza per entrare:

- push EBP
- mov EBP, ESP
- sub ESP, n

è stata riassunta dai progettisti Intel con enter, che però non viene usata da compilatori per C/C++, in quanto fa qualcosa in più: riceve come primo argomento n, ma poi c'è un secondo argomento che riguarda l'annidamento della funzione m. Ma in C/C++ non è possibile creare funzioni annidate, quindi m sarebbe sempre 0, C non lo permette perché diventa più difficile la gestione dello stack.
Esiste però la leave, che è l'opposto della enter:

- mov ESP EBP
- pop EBP

poi c'è la ret. Le istruzioni della leave, copiamo EBP in ESP e quindi ESP punta di nuovo dove puntava EBP, per cui lo stack può essere stato sporcato quanto si voleva, ma questo lo ri-bilancia, ora la pop rimette in EBP il valore che aveva nell'istruzione chiamante, quindi ESP punta all'h, ovvero all'indirizzo di ritorno. Nelle funzioni Assembler c'è sempre un **prologo** ed un **epilogo**, che sono enter e leave.
Anche le librerie di Win hanno le funzioni di base comune, come la puts, l'argomento viene messo nello stack con mov: gcc decide, per ottimizzare, di allocare direttamente i 16 byte[1] per passare i parametri e poi dereferenzia lo stack pointer, Ghidra aiuta perché all'inizio della funzione dice come viene usata la funzione:

- cosa ritorna
- se riceve valori di input

i riferimenti di Ghidra sono fatti "ad alto livello" ed è possibile cambiarli, cosa che può aiutare durante l'analisi.
Spesso i programmi Intel vengono usati in maniera da non usare il base pointer, quindi non c'è EBP per puntare alla base dei frame e quindi ogni volta che viene modificato lo stack il compilatore rifà tutti i conti per risparmiare sul registro EBP ed in questo Ghidra è ancora più importante perché fa in conti per noi.

1.3 Come costrutti ad alto livello vengono tradotti in Assembler

Entry non è quello che il programmatore ha chiamato "main" nel programma C, prima bisogna eseguire il codice per creare l'ambiente di esecuzione per il processo da eseguire. Una delle funzioni usate sarà main, ma non è detto che verrà invocata da una chiamata in entry, è possibile che ci sia solo l'indirizzo di main che verrà poi passata ad una qualche funzione per invocarla.

1.3.1 Trovare il main

Per cercare di trovare il main, se abbiamo un'idea di cosa faccia possiamo provare a risalire all'indietro: vediamo le chiamate di sistema del programma, che si trovano nella sezione **imports**, dove si riassume tutto ciò che verrà invocato in una DLL. Possiamo trovare delle librerie con altre funzioni , tra cui magari printf e possiamo quindi chiedere a Ghidra di saltare al thunk di printf o anche saltare a tutti in punti dove viene chiamata printf. Generalmente la printf è usata dal programmatore, non da una libreria, e quindi possiamo risalire all'indietro finché non troviamo il main: chiediamo sempre con X chi invoca la funzione. Il main può avere l'aria di una funzione che inizializza l'ambiente, una volta scoperto quella che penso sia il main occorre rinominarla per non dimenticarsi dove è.
Le convenzioni di uso delle funzioni vogliono che una funzione abbia diritto a "sporcare" determinati registri, ma non ha diritto a sporcarne altri, quindi è necessario fare la push ad esempio

[1] può servire per ottimizzare il codice, per esempio per individuare sempre l'inizio di una linea di cache

di EDI, EDX etc e di ripristinarne il valore con una POP prima di fare la return (lo stack sarà bilanciato). Per capire come funzionano programmi complessi, uno dei punti di partenza è quello di cominciare a capire quali sono le strutture di dati usate dalle funzioni, perché poi magari queste guidano nella comprensione di cosa il programma fa.
Ci sono una serie di istruzioni dell'Intel che permettono di automatizzare le funzioni sulle stringhe, come ad esempio cercare uno 0 nella stringa etc... ed anche se l'istruzione è di soli due byte, coinvolge diversi registri. Queste vanno viste e comprese dal manuale.
Nel caso in cui non dovesse funzionare il risalire a ritroso, si può vedere dal grafo:
trovare cose del tipo [DAT_qualcosa] vuol dire leggere una variabile globale dalla memoria per caricarlo in un registro. Quando nel programma ci sono vari salti, Ghidra permette di vedere un grafo della singola funzione (diverso da quello di prima): Ghidra organizza il codice in blocchi, dove si entra dalla cima e si esce dal fondo, è consigliabile mettere delle etichette anche su costrutti for/while quando si identificano nel flusso di esecuzione
La lea sta per load effective address, ovvero chiedere di mettere in un registro il valore usando un elemento dell'architettura del processore dedicato, e non la classica ALU. È un modo quindi, ad esempio, di fare una operazione aritmetica senza usare la movl. Ghidra permette, mano a mano che si fa reversing, di commentare cosa si scopre qualcosa e magari si possono mettere dei commenti speciali che magari cambiano stato nel tempo.
Più avanti, è possibile che si passi dal fare operazioni sui registri a 32 bit a registri a 16 bit, quindi usare dei nomi che ricordino che sono degli short in C.
Conoscendo il compilatore, sappiamo che la chiamata la main avviene in un certo punto, quindi si va a cerare quel punto preciso, ma questo dipende da vari fattori come la libreria C usata ed il compilatore, quindi trovarlo è un aspetto cruciale.
Vedendo il function call graph, possiamo vedere i blocchi di codice, ovvero pezzi di codice eseguiti in sequenza, e possiamo analizzare i diversi blocchi per capire cosa fanno

1.3.2 Capire le funzioni interne

Cerchiamo quindi di capire cosa fanno le chiamate interne al main. Le cose più interessanti sembrano le chiamate a funzione generiche, a cui conviene ad associare dei nomi, anche se non simbolici, per ricordarsi che le si sta analizzando. Cerchiamo di capire, per ogni chiamata a funzione, quali e quanti sono gli argomenti che vengono passati come input. Fatto questo, si cerca di capire cosa fa la funzione chiamata, la prima cosa fondamentale da fare è ricostruire le strutture dati del programma. Possiamo provare ad usare un decompiler per capire cosa fa la funzione, ma il rischio è che questo fallisca nel capire quali sono i tipi di dato delle variabili in gioco, quindi rischia di classificare come short qualcosa che non lo sia. Inoltre, in Ghidra, tutte le variabili locali che si sovrappongono sulla stessa zona di memoria, mette degli "_", e l'obiettivo è non avere underscore in quanto vuol dire aver identificato i tipi delle variabili.
Nel chiamato troviamo sempre prima il prologo, con cui si salvano i registri di cui si vuole salvare lo stato, inoltre la struttura del codice non è sempre uguale a quella pensata dal programmatore in quanto il compilatore fa una serie di ottimizzazioni interne.
Trovando delle operazioni che sfruttano delle variabili, se capiamo il tipo della variabile perché ad esempio vi viene caricata una word (quindi è un intero), va subito segnato:

- si da un nome che ricordi il tipo di dato
- doppio click, si va nel listato di Ghidra dei dati, per cambiare il tipo della variabile (tasto dx, cambiare tipo di dato)

Ora, alcuni "?" di Ghidra sono risolti, il lavoro più lungo sta proprio nel mettere nelle strutture dati cosa si capisce dal codice macchina.
Può accadere che Ghidra non indichi, ad esempio in una mov, dword ptr quando si manipola un registro a 32 bit. Ma questo perché le due mov, ovvero questa e quella che esplicita dword ptr, abbino dei codici operativi diversi, ma questo non è un problema.
SAR: scorrimento a destra, che però prevede il rientro del segno: quindi se abbiamo 1xxx, otteniamo 11xx quando shiftato di destra. Si può ottenere questo tipo di istruzione in C quando abbiamo una variabile int v, a cui segue un v >>1. SHR fa la stessa cosa ma senza considerare il segno. In base al fatto che si vada a fare la mov di una word da un registro int, fa capire che si prendono i bit meno significativi, quindi si sta gestendo un long, questo si può fare per ogni variabile che si incontra nel codice, tenendo presente alcune cose come ad esempio il fatto che in Ghidra un char è inteso come un carattere di una stringa, quindi se c'è un carattere usato in una somma è un tipo byte. Se un dato intero a 32 bit viene mappato in un int o in un long del C dipende dal compilatore, quindi generalmente si tende ad usare dei tipi di dato che includano nella definizione il tipo; in C ad esempio, un modo per essere agnostici dal SO o dal compilatore è usare la stddef.h in modo da aumentare la portabilità del codice.
Ora, dopo aver capito i tipi di dato, è possibile rivedere il de-compilato, che ora è più capibile di prima, ma comunque non è detto che si riescano a capire le dimensioni in gioco, mentre dall'assembler è possibile farlo.
Se abbiamo degli if nel codice, ad esempio con dei confronti fra interi, si usano XOR e JNZ per capire se saltare, in modo che si decida se prendere il blocco dell'if o meno.
Anche per ai for loop o while loop conviene dare dei nomi per rimarcare che sono dei loop, possiamo riuscire a trarre informazioni anche dal de-compilato, anche se partendo dalle istruzioni macchina si ottiene spesso del codice più lungo di quello prodotto dal programmatore.
Se Ghidra non riesce a riconoscere del codice o dei dati, li sostituisce con '??', a quel punto si può provare a disassemblare le zone che si crede siano codice per vedere se le istruzioni macchina sono realmente presenti o meno; può accadere che il compilatore abbia sbagliato per via delle offuscazioni utilizzate da chi scrive il malware.

Jump table

Nel caso di switch-case, nel codice macchina si ha la presenza delle jump table: ogni switch ha un certo numero di casi definiti, più un deafult opzionale. La prima cosa che viene fatta quando un compilatore compila uno switch è confrontare il valore dell'espressione dello switch col totale dei casi, se è maggiore salta direttamente al default case. Se invece il valore dell'espressione rientra nei casi, si usa la jump table ovvero si salta alla locazione della cella indicizzata dall'indice *4, sommando poi la base della jump table, ovvero la base della tabella che contiene gli indirizzi dove si salta. Quindi, ogni entry della jump table è la base per l'indirizzo di salto dei diversi case, a cui sommiamo l'indice dello switch, dato quindi dal valore della variabile, moltiplicato per 4. Può accadere che un compilatore ottimizzante vada a sostituire ad una function call una funzione inline, ovvero direttamente del codice macchina, cosa che avviene per motivi di efficienza (a meno che non sia disabilitato per default).

Chapter 2

Lezione 6

2.1 Struct, liste collegate ed utilities di Ghidra

Può accadere spesso che il compilatore estenda una chiamata a funzione all'interno del chiamante, per motivi di ottimizzazione.
Abbiamo una struct in cui ci sono una serie di assegnazioni fra due istanze di struttura diverse, che però vengono mappate in assembly in maniera diversa:

- vediamo delle mov tra memoria e registro
- vediamo delle moltiplicazione
- etc...

quando la struttura è assegnata ad una variabile globale o automatica, il compilatore implementa il codice delle strutture conoscendo gli indirizzi delle due istanze di struct, quindi i riferimenti ai campi delle due strutture vengono automaticamente risolte. Per arrivare al campo, occorre fare la somma della base dell'indirizzo della struttura + l'offset per arrivare al campo, quindi secondo il problema è quello di ricostruire il sorgente, un codice assebmly di questo tipo aiuta poco. Potremmo sospettare che sia una struct perché gli accessi in memoria sono in zone contigue, ma il problema è che fin ora non possiamo sapere se questo sia stato un caso o se siano davvero accessi ad una struttura. Si può però capire andando avanti nell'analisi, studiando ancora le funzioni chiamate nel main.

Ad esempio, possiamo trovare un accesso ad una memoria che abbiamo trasformato prima: vediamo l'accesso ad un byte che abbiamo visto prima, ma a questo punto potrebbe essere una contraddizione perché qui vediamo che il byte viene caricato in un registro a 4 byte. Vediamo l'uso del registro dove viene caricato il presunto byte: può capitare di trovare dei no-op con delle istruzioni come la lea di un registro in se stesso, utili per allineare la memoria ad un indirizzo multiplo di una potenza del 2, come 16: può darsi che ci sia un'istruzione di salto, quindi si allinea la memoria ad una linea di cache per motivi di ottimizzazione.

Il codice usa l'indirizzo della variabile byte che abbiamo identificato per andare a spiazzarsi, ne usa l'indirizzo perché la mov senza "[]" prende l'indirizzo, non il contenuto del pointer. Quindi le istruzioni successive usano la base dell'indirizzo con degli offset, questo è indicativo del fatto che può essere coinvolta una struttura: ogni volta che si usa un indirizzo di abse e gli accessi non sono diretti, ma usano degli offset, questi offset sono relativi ai campi della struttura.

Altra nota importante è che non è vero che la somma delle dimensioni dei campi determina la dimensione della struttura, perché il compilatore può fare padding di byte vuoti per motivi di efficienza, perché ad esempio attaccando il campo a quello precedente si finisce in indirizzi di memoria non multipli di 4 byte, per si perde in prestazioni. Quindi, MAI ASSUMERE che se ci sono dei byte in una struttura hanno significato, perché magari è solo padding.

A questo punto, dobbiamo riportare l'informazione del fatto di aver trovato una struct dentro Ghidra: andando nella zona dati dove è presente la struct, possiamo selezionare una certa quantità di byte che ho capito far parte della struttura, usando il tasto dx del mouse possiamo selezionare "data → create structure". Ghidra chiede se la struttura è ricorrente del SO, in modo da far vedere in un menù tutte le strutture che hanno un numero di campi pari a quelli selezionati, però bisogna aver "indovinato" il numero esatto dei campi della struct. Non abbiamo motivo di pensare che la struttura sia del SO. Dando un nome generico, Ghidra collassa tutti i campi in una struct.

A questo punto, il codice non è cambiato molto, invece di avere DAT_qualcosa, viene aggiunto il fatto che si accede ad un campo di una struct, ma i campi vanno definiti: per farlo, occorre tornare nella struct e fare l'edit del tipo di dato, in base a come il codice accede ai dati possiamo capire il tipo dei campi.
Ci rendiamo poi conto che, tramite il debugger, c'è l'assegnazione di un campo della struct ad un'altra struct dello stesso tipo, quindi pensiamo ad una lista collegata a cui si sta assegnando il puntatore al campo next; anche questo va segnato, dandone anche il tipo che indichiamo come dword (in quanto è un pointer ??).

Nel main originale, troviamo che alcuni riferimenti sono risolti ed altri no, questo perché nel codice macchina non ci sono altre istruzioni che suggeriscono ad altri accessi ai campi delle strutture: possiamo capirlo perché ricostruiamo che gli offset relativi sembrano essere gli stessi in alcuni casi, magari perché viene nuovamente usato il singolo byte come indirizzo di partenza, e quindi andiamo a selezionare la zona dei dati in corrispondenza della struct; avendo precedentemente memorizzato la struttura, se i campi selezionati sono gli stessi, Ghidra la indica tra i suggerimenti. Lavorare riportando dentro Ghidra le informazioni è cruciale, altrimenti non se ne viene a capo, in generale il data type manager elenca tutti i tipi di dati che sono stati inseriti o auto-appresi analizzando il file. Questa è una delle cose più importanti, perché quando si fa reversing, si va a caccia delle strutture dati (cit*).

Fra le cose più importanti ci sono le stringhe del programma: tra i vari motori di analisi che Ghidra fa quando carica un eseguibile, ce n'è uno che cerca le occorrenze delle stringhe in modo da provare ad arrivare ad informazioni auto-referenziali. Se vediamo ad esempio una stringa che è un messaggio di errore, sappiamo che la stringa è associata ad una funzione di errore, ed inoltre che la stringa è associata ad tipo di errore che è avvenuto.
Altre stringhe sono importanti perché fanno parte del meccanismo con cui il programma è collegato alle DLL, uno dei modi fa il collegamento usando il nome della funzione da invocare, quindi abbiamo una stringa. Supponiamo di sapere che il programmatore abbia usato la printf, trovandola fra le stringe ne abbiamo una conferma e possiamo clickarvi per approdare nel punto dell'eseguibile dove è stata definita la stringa. Sono utili quando associate al meccanismo dei riferimenti: se prendiamo dei commenti, alcuni di questi sono delle **xref** che indicano in che

punto del codice si trova un riferimento a quel simbolo. Guardando una funzione, vorremmo avere un modo di correlare la funzione stessa a tutti i punti in cui appare, stessa cosa per le variabili: Ghidra dice in automatico dove vengono referenziate, nella sezione commenti, e permette di ricostruire cosa è un certo tipo di dati in base a come questi vengono usati.
Spesso Ghidra non è in grado di risolvere dei riferimenti, in linea generale è possibile aggiungere quelli che non riesce a calcolare automaticamente, i riferimenti sono gestiti come un grafo orientato e quindi aggiungerne uno vuol dire aggiungere un arco orientato.

2.2 Traduzione in Assembler di programmi scritti con linguaggi OO

Il reversing di programmi Java è completamente diverso da quello visto fin ora: l'eseguibile del Java è il bytecode che verrà poi eseguito da una VM, siccome il bytecode nasce per Java è molto vicino a Java stesso. Il reversing non è molto complesso, tranne nel caso in cui il codice viene offuscato, andando a nascondere le stringhe, ci sono una serie di de/offuscatori del codice, i programmi Java sono importanti perché Java è il linguaggio di punta con cui vengono sviluppate le app per Android.
Si parla del reversing di programmi OO perché chi scrive il malware si appoggia a compilatori comuni, il più comune del mondo Microsoft prevede l'utilizzo di librerie OO, quindi molto malware in circolazione è un programma ad oggetti, quindi occorre capire almeno le problematiche legate a questo.

Un programma C++ è completamente differente ad uno C, anche in termine di codice macchina ottenuto, il C++ è molto più sofisticato e quindi il codice macchina è molto più complesso. Nella programmazione ad oggetti si parla di

- polimorfismo, ovvero la capacità a run time di una stessa funzione di avere comportamenti differenti ai parametri passati, per farlo il nome codifica i parametri che vengono passati. Il problema è solo a livello di reversing, non di programmazione;
- ereditarietà: un oggetto viene esteso nelle funzionalità da un altro, quindi possiamo definire delle strutture dati partendo da strutture esistenti e derivandone altre proprietà

quindi il livello di complessità dell'analisi di un programma ad oggetti cresce parecchio.

Vediamo il reversing di un programma C++: una classe non è altro che una struttura del C con dei campi che sono puntatori a metodi della classe, quindi non è necessario usare un linguaggio OO per programmare OO, lo si fa perché è più pratico. Tutto ciò che un programma fa in termini di OO può essere fatto con un linguaggio non OO, ma è meglio tagliarsi il cazzo che farlo.
Comunque, programmare ad oggetti è un paradigma, non è obbligatorio usare un linguaggio OO, **il classico esempio è quello del kernel Linux**, dove altrimenti si produrrebbe del codice molto più pesante. Un altro aspetto della programmazione OO è l'incapsulamento, ovvero la definizione di metodi e attributi privati o pubblici o protected.

Abbiamo poi il **costruttore**, ovvero un metodo che viene eseguito ogni volta che un nuovo oggetto della classe viene eseguito. Alcuni metodi possono essere virtuali, ovvero è possibile

ridefinirli da parte di classi che ereditano la classe padre, se la classe figlia definisce dei metodi con lo stesso nome del padre va a farne una ridefinizione. Ci sono dei casi in cui le funzioni virtual sono "pure", ovvero simili ad i metodi "abstract" del Java, per cui il metodo va implementato nelle classi figlie e chiamato solo dalle classi figlie.

Il punto è capire come sono fatte le strutture dati del programma, supponiamo di istanziare una sotto-classe assegnandola ad una variabile di tipo della classe padre: quando viene creato un nuovo oggetto, viene allocata della memoria nell'heap, in memoria troveremo lo spazio per tutti i campi della classe, ma anche per i metodi. Per avere spazio per i metodi, la struttura ha come primo campo un puntatore che punta alla vtable associata alla sotto-classe, NON all'oggetto, la vtable contiene gli indirizzi dei metodi definiti nella sotto-classe ma anche quelli ereditati dal padre. Esiste anche una vtable dell'oggetto padre, che avrà solo gli indirizzi dei metodi definiti nel padre. I campi non sono codice, ma ancora puntatori a funzioni, che punteranno alle implementazioni delle funzioni stesse, nel caso delle funzioni puramente virtuali

- nel padre punta ad un errore
- nel figlio punta all'implementazione di quella funzione

a seconda del fatto che la funzione venga o meno ridefinita, si punterà a funzioni uguali a quelle del padre o a funzioni ri-definite.

Il problema è che tutto ciò è dinamico: possiamo considerare il parametro associata all'istanza di classe figlia, tramite un cast, come istanza della classe padre e quindi cambia la vtable da considerare. Per questo, c'è una relazione fra le vtable, che vengono accorpate per motivi di ottimizzazione dal compilatore, in ogni caso il messaggio è che ricostruire il funzionamento di un compilato da un programma ad oggetti si parte dalle vtable.

Tutti i metodi possono avere degli argomenti, nel C++ ce n'è sempre uno implicito, il **this**, che punta l'istanza della classe ovvero all'oggetto per il quale quel metodo virtuale viene chiamato. Se abbiamo due istanze, una di sotto-classe ed una di padre, con this capiamo quale delle due è quella per cui invocare un metodo comune. Il this deve essere presente a livello Assembler, questo aiuta a capire che il programma è OO e che al funzione è il metodo della classe, perché si usa sempre lo stesso meccanismo di passaggio del parametro, che però cambia da compilatore a compilatore:

- nel caso del Microsoft cazzo vattelappesca il this è passato nel registro exc;
- in g++ o altro, this viene sempre considerato il primo parametro nascosto di qualunque funzione. Sostanzialmente il puntatore viene messo sulla cima dello stack quando si entra nella funzione

a 64 bit le cose cambiano ancora, si usano differenti registri ma non ce ne sbatte un cazzo perché approfondiamo solo 32 bit sennò ce serve un anno sano solo per questo esame.

Chapter 3

Lezione 7

3.1 Funzionamento delle applicazioni Windows

Cominciamo ad esaminare nel dettaglio come funziona un'applicazione Windows. Questo perché per fare il reversing dell'applicazione, occorre sapere come è fatta, inoltre gran parte del malware è Windows ed usa API di Windows. Proviamo a capire la struttura dell'applicazione, il modo migliore è aprire un app che è una singola finestra vuota: effettivamente, la finestra è già funzionale, ovvero contiene tutto ciò che caratterizza la finestra di Windows:

- muovere
- minimizzare e massimizzare
- menù gestito dal SO

Quindi la finestra racchiude il minimo indispensabile per implementare la finestra.
Proviamo a ricostruirne il funzionamento a partire da Ghidra: convenzionalmente, le applicazioni grafiche di Windows hanno come programma principale il "WinMain" e non il main, non è obbligatorio che la funzione iniziale si chiami "WinMain", perché dipende dal framework dove è stato scritto. Generalmente, le applicazioni Windows grafiche non vengono scritte "a mano", ma usando dei framework specifici ed nel mezzo ci sono degli strati di software che complicano la vita a chi fa analisi.
Altra questione, come fare per cercare la documentazione:

- libri che spiegano come si programmano le applicazioni;
- tradizionalmente, cercando il nome della funzione + " msdn", si ha come primo link la documentazione ufficiale di Microsoft, per ciò che p documentato.

Può accadere che, compilando con Microsoft VS++, ci siano due possibili modi di codificare i caratteri

- ASCII
- UNICODE, per supportare i caratteri speciali delle diverse nazioni. UNICODE ha diverse codifiche, più ricche dell'ASCII, considerando 2 byte per carattere

Scegliendo una o l'altra, è un problema se si riceve come input una stringa: WinMain è il punto in ingresso specifico per il formato ASCII, perché uno dei parametri è una stringa ASCII, quindi se vogliamo supportare l'UNICODE c'è wWinMain. Qualunque funzione della libreria DLL che prende come input una stringa codificata in formato ASCII ha due versioni diverse, una delle due è per UNICODE. A livello di programmazione è tutto uguale, sarà il compilatore a scegliere la versione giusta in base al formato della stringa, ma la versione UNICODE non è documentata, quindi poi sta a chi fa reversing capire.
La WinMain usa 4 parametri, due dei quali sono HINSTANCE, che sono dei typedef per degli handler ed oscuri al programmatore

- HINSTSNCE hInstance: il processo corrente;
- HINSTSNCE prevInstance: usato per motivi di portabilità, non si usa più;
- LPSTR lpCmdLine: linea comando con cui è stato invocato il programma, con solo i parametri e senza il nome dell'eseguibile
- int nShowCommand, flag che indica come deve essere visualizzata inizialmente la finestra

3.1.1 Reversing con Ghidra

Il lavoro di reversing consiste quindi nel cercare l'API e leggere la documentazione, e capite le strutture di dati da passare al programma, capiamo anche quali sono le strutture dati del programma in analisi.
Andando in Ghidra, troviamo cosa ha capito dei parametri nello stack: quando identifichiamo una API di Windows di cui Ghidra non ha riconosciuto i parametri, vado ad editarlo io mettendo anche il nome ufficiale dalla documentazione.
La prima API invocata è la loadIcon, che andiamo a cercare in documentazione e scopriamo che le applicazioni Windows oltre ai dati, usano anche delle risorse: le risorse possono essere icone, etc... Sono indipendenti dal codice e dai dati, tant'è vero che possiamo mantenere il programma così com'è modificando solo le risorse (ricorda Mobile Programming, quando modificavi le stringhe definite nel file per aggiungere quelle nazionali etc...). Scopriamo poi che tra le opzioni c'è la scelta dell'icona da usare, che in Ghidra è un valore numerico, ma possiamo associare la marco a mano tramite tasto destro, "Equate" in modo da memorizzare la macro.
Altra cosa importante: le chiamate a DLL usano la convenzione 2, la ret toglie gli argomenti dallo stack al ritorno (ret 8 per dire togli 8 byte, ovvero fare add ESP di 8). GCC non segue questa convenzione, quindi appena l'API torna, GCC fa la sub dei byte aggiunti dall'API per ristabilire l'ordine.

RegisterClassex funzione importante, che registra una classe finestra: ogni finestra corrisponde ad una classe, quindi prima si crea la classe e poi si creano le istanze della finestra con la CreateWindow. La funzione prende un solo parametro, che è puntatore ad una struct. Il parametro passato è in EAX, viene impostato in EAX, usando l'indirizzo di local_5c, ma quindi sappiamo che quella è un puntatore alla struttura che la funzione di aspetta. Mettiamo come tipo il nome della struttura, scrivendolo oppure cercandolo fra quelle note a Ghidra.
Quindi, per ogni argomento delle funzioni, cerchiamo nella documentazione a cosa servono e quali sono (se ci sono) i possibili valori; inoltre, nell'equate è possibile trovare in automatico la corrispondenza fra macro e valore numerico (se presente), filtrano ad esempio per alcune delle

lettere.

Infine, la ShowWindow fa apparire la finestra, in base al secondo argomento costante si decide come far apparire la finestra, ora occorre capire cone fa Windows a gestire la finestra

Funzionamento della finestra

Ogni MainWin fa un loop while infinito in cui vengono eseguite operazioni fondamentali: supponiamo di avere una entità SO Windows, il programma che gestisce tutti i programmi Windows, ed il programma in analisi. Abbiamo definito una WinMain che è un ciclo senza fine, ma come fare per personalizzare la finestra? Tutta la gestione delle applicazione grafica in Windows è basata sugli **eventi** o **messaggi** che vengono ricevuti dal SO. Il messaggio arriva nel MessageLoop, perché magari l'utente muove il mouse o clicka una finestra etc...
L'applicazione leggere un messaggio con GetMessage, la seconda API è Translate: ad esempio, potrebbe tradurre lo shortcut dalla tastiera in un altro evento che vi corrisponde, quindi trasforma il messaggio prima di gestirlo.
Il messaggio viene gestito usando DispatchMessage, perché il controllo torna al SO: quando si registra una classe, in uno dei campi c'è l'indirizzo della funzione WindowProc. WinMain è globale per l'applicazione, ma poi quando viene fatto il dispatch, il SO lo recapita alla WindowProc, ovvero esegue la WindowProc collegata alla finestra che ha dato origine al messaggio.
Il Dispatch fa in modo che il messaggio arrivi alla WindowProc, ovvero la procedura della finestra che lo gestisce e che ha il codice che specializza la finestra rispetto a tutte le altre.
C'è una WindowProc per ogni finestra, quindi l'analisi si continua da lì.

WindowProc

Siccome la call alla WindowProc non è stata vista da Ghidra, è importante dirgli di trattarla come una funzione (tasto dx, qualcosa con function). La WindowProc ha un particolare prototipo, che va riportato su Ghidra:

- ha un risultato
- prende dei parametri
- etc...

tutti questi aspetti vanno editati in Ghidra. Dopo di che, vediamo cosa deve fare la funzione: Windows definisce una gestione di default per tutti i messaggi: quando si fa resize della finestra, funziona perché è stato Windows di default ad aggiungere il codice, così come per chiuderla etc... La WindowProc deve solo intercettare i messaggi per i quali vuole discostarsi dal default, vedendola col decompilatore troviamo delle compare con dei codici numerici, anche in questo caso andiamo ad associare con equate. Per dire a Windows che i messaggi di default vanno gestiti, basta usare la API DefWindowProc, passando gli stessi argomenti avuti come input, in questo caso reagiamo a WM_DESTROY, usando la API PostQuitMessage per fare una chiusura forzata dell'applicazione.
C'è una jump ad una label che Ghidra non risolve, ma possiamo associargli un nome, è il wrapper per risolvere le DLL. Quando la class è seguita da una return, se non va modificato lo stack fra la call e la return, tanto vale fare una jump, DefWindowProc ha lo stesso numero di parametri, quindi tanto vale non fare call sub e return ma fare direttamente il jump, **quindi call f, ret $\leftrightarrow$ jump f**

Chapter 4

Lezione 8

4.1 Reversing di applicazione

Applicazione Windows, del tipo "capture the flag". Abbiamo un programma con una limitazione sul tempo per cui è aperto, quindi dobbiamo capire come fare a rimuovere questa limitazione mediante analisi statica del codice.
Sappiamo già che tutti i programma Windows sono costruiti intorno al MessageLoop, quindi se lo troviamo riusciamo anche ad individuare il WinMain. Andiamo nelle funzioni e cerchiamo le tre funzioni che compongono il loop. Individuata la WinMain, tanto vale rinominare e impostare i tipi dei parametri ed il tipo di ritorno. Riusciamo ora a trovare anche la WindowProc, ovvero il processo associato alla finestra, sappiamo che uno dei campi della struct è l'indirizzo della WindowProc, quindi andiamo alla label contenuta nella variabile ed impostiamo in Ghidra il fatto che inizia una funzione, anche qui impostiamo il prototipo della funzione.
Non dobbiamo mai scordare l'obiettivo: in questo caso, il nostro obiettivo è disattivare il TO dell'applicazione, quindi evitiamo tutti i messaggi che non ci interessano, come tutto ciò che appare come pop-up nella finestra. Cerchiamo quindi di capire che messaggi gestisce la procedura:

- possiamo usare il decompilatore
- capire dall'assembler, in particolare dal grafo della funzione, cosa sta succedendos

Dal grafo della funzione, cerchiamo switch o una serie di if per la gestione dei messaggi: notiamo che in ebx viene messo il tipo di messaggio, per poi confrontarlo con valore 5, andiamo quindi a targare il messaggio con la macro corretta (nel caso dei messaggi, iniziano con "WM"). Capiamo che il salto avviene sul blocco che fa il resize della finestra, quindi non è interessante. Andiamo avanti

- abbiamo poi una JBE, quindi se ebx è ≤ 5, saltiamo in un altra parte di if, dove c'è un confronto col valore 1. Questo viene chiamato quando si crea la finestra
- altri confronti con degli altri flag che hanno dei significati che sono ricercabili dalla documentazione
- troviamo il codice che corrisponde a "WM_COMMAND", che viene mandato quando l'utente seleziona o una voce di menù oppure un controllo manda una notifica alla sua finestra genitore. L'edit box ad esempio, è una finestra (ovvero l'elemento della finestra è

una finestra), quindi quando succede qualcosa manda una notifica alla finestra genitore. È improbabile che sia questo quello che cerchiamo

Dopo aver visto tutti i messaggi, ci chiediamo dove può essere il punto che ci interessa: l'applicazione parte e dice che si chiuderà dopo un certo tempo, serve quindi aver impostato un meccanismo che faccia il conto dei secondi e la faccia chiudere. Vediamo il codice del WM_COMMAND, sembra più "esotico" degli altri.

4.1.1 WM_COMMAND

Abbiamo una label che rinominiamo come handle_command_log: si lavora sul registro ESI, bisogna capire chi è l'ultima volta ha impostato il valore di ESI, vediamo poi dalla documentazione che significato ha. Il valore di wParam dipende da chi ha generato il messaggio, vediamo per esclusione che è stato impostato da un qualcosa che ha definito l'applicazione. A questo punto, andiamo a confrontare il valore di EDI con quello contenuto in una certa area di memoria, se sono diversi si esce. Cerchiamo di ricostruire il perché si legge il parametro sullo stack local_b8. Vediamo fra i riferimenti dove viene scritto il valore e notiamo che avviene una volta sola. Vediamo che ci viene scritto il valore EAX, sappiamo che è il registro accumulatore dove viene restituito il valore di una chiamata a funzione. Troviamo che la prima "occorrenza" è la chiamata ad una funzione **GetWindowLongA**. La funzione restituisce una informazione legata alla finestra, inoltre restituisce una word legata ad una memoria aggiuntiva allocata durante la creazione della finestra.
In base al parametro passato, possiamo avere diverse opzioni, e scopriamo quale valore viene usato: il flag è un dato passato dal programmatore. Abbiamo quindi capito che la procedura della finestra legge un valore associato alla finestra dal programmatore, ma come fa a metterlo nella finestra?

4.1.2 Create Window

Dalla documentazione cerchiamo da chi viene impostato il valore, e scopriamo da chi viene impostato, ed è la **SetWindowLong**e vediamo dal func call graph che viene chiamata dalla WindowProcedure: il momento più sensato per fare il set del valore è quando viene gestito l'evento di creazione della finestra. Vediamo che la funzione ha 3 parametri, quindi vediamo che viene impostato ad EDI che era lParam e ci chiediamo cosa è lParam per la wm_create. Scopriamo che è un puntatore ad una struttura che contiene informazioni relative alla finestra che viene creata, e vediamo che della struct viene passato il primo parametro, dalla documentazione non si trova il valore che viene passato, si torna al WinMain per vedere come è stato specificato. Vediamo che il parametro viene impostato come valore di ritorno di una funzione: la funzione crea una struct ed al suo interno ci sono delle altre chiamate, alcune sono assurde quindi le skippiamo a pie pari. Ci focalizziamo sull'inizializzazione delle strutture di dati, che sono quello che dobbiamo seguire; abbiamo anche visto che il valore restituito dalla funzione sarà passato il lParam: viene restituito l'indirizzo ad una singola variabile nel .bss inizializzato a 0, anche se ci sono diverse aree di memoria, quindi può essere una struct.
Il valore local_8 è confrontato col contenuto del registro EDX + un certo offset: local_b8 è l'indirizzo locale della ipotetica struttura che viene inizializzata dopo, quindi EDI viene confrontato con uno dei campi della struttura, perché l'offset viene preso rispetto ad un campo della struttura. Creiamo una nuova struttura in Ghidra, tramite "Window, new data", siccome non sappiamo di quanti campi è fatta, ma sappiamo che si accede ad un campo ad offset 184

copiandoci un pointer a dword, mettiamo un primo campo di 184 byte e poi il nostro campo di 4 byte ad offset 184.
Man mano che scopriamo quali altri offset della struttura vengono acceduti, si aggiorna la struttura dati definita. È vero che la Create non è direttamente legata la timer, ma seguendo i dati scopriamo che usa la struttura dati e ci da delle informazioni su come è fatta internamente nei campi. Arriviamo a SetTimer, dopo aver percorso tutti i campi della struttura

Chapter 5

Lezione 9

5.1 Continuo dell'analisi - InitDS

Completamento della versione demo del programma che spegne il PC. Quando si analizza un codice, se si può bisogna seguire i dati, non le istruzioni perché su queste si può perdere. Torniamo sulla funzione **InitDS**, che inizializzava una struttura di dati che però non è ancora completa. La prima istruzione che troviamo è una mov di 4 byte nella struttura, quindi possiamo aggiungere l'informazione sui campi della struttura. Per capire quando Ghidra scrive qualcosa nello stack, basta vedere se la locazione dello stack sarà sia scritta che letta: se avviene ciò, le variabili saranno locali, se vengono solamente scritti sarà solo un passaggio di parametri; conviene sempre seguire la denominazione degli argomenti come "fn_arg{i}". Anche per i campi delle struct di cui non si sa dire cosa siano, conviene nella finestra per la gestione della struttura scrivere nel nome qualcosa che ricordi a cosa serve. Troviamo, andando avanti una CALL ad una funzione: è abbastanza evidente che venga passato come primo argomento uno dei campi, ma dopo un wrapper, c'è la chiamata alla "vera" funzione che sembra molto complessa. Quindi piuttosto che capire cosa fa la funzione, conviene vedere quali altri punti invocano la funzione. Troviamo una funzione contenente una stringa di formato, del tipo "%2 stringa", quindi ci ricorda il formattatore della printf, usata anche nella famiglia sprintf e che la funzione trovata sia una di quella famiglia lo suggeriscono anche gli argomenti passati:

- abbiamo come secondo argomento una dimensione intera, e ci ricordiamo che è la snprintf ad avere come parametro un intero che è la taglia della stringa
- poi ci sono gli altri parametri

Quindi rinominiamo la funzione precedente con snprintf, ed otteniamo informazioni sulla struttura dati di partenza: questo può accadere perché, a differenza di funzioni di DLL, la snprintf viene linkata a compile time, aggiungendo direttamente il codice della funzione e quindi nel file abbiamo un indirizzo, per cui Ghidra non riesce a riconoscerla.
Riuscire a distinguere la funzione di libreria linkata staticamente da una scritta da un programmatore si ottiene con la pratica, ma è comunque una ipotesi perché lo scopo è comunque cercare di capire cosa fa la funzione.
C'è poi una seconda stringa esadecimale

5.2 Ultime funzioni

Torniamo a vedere cosa avveniva nella funzione **handle_create_message**, è possibile aggiungere a Ghidra dei riferimenti alla struttura dati che non sono stati chiariti: mettiamo mediante "set reference" (qualcosa del genere) e Ghidra metterà una freccia come riferimento (IMPORTANTE: ecco cosa vuol dire la freccia?) Scopriamo quindi che vengono usati 5 handle, quindi nella struttura conviene raggruppare i 5 handle in un array, quindi possiamo spostarci sulla prossima funzione: vengono fatte una serie di operazioni. Abbiamo un operazione in aritmetica intera: moltiplico e divido un valore per 60, ma se semplifico ottengono un risultato differente, in quanto ci sono di mezzo delle approssimazioni, la formula serve a calcolare il numero di secondi rimanenti. Ora rimane da capire in che unità di misura si sta contando il tempo

- 1800 secondi sono 30 minuti, che è il valore di inizio
- il contatore dice quanto tempo è passato

possiamo quindi cambiare i campi delle struct con dei nomi più significativi. Serve un meccanismo che misura il passare del tempo, sapendone l'unità di misura possiamo sapere quanto tempo manca.
Dopo che si crede di aver capito il funzionamento, occorre cercare di verificare le conclusioni di cosa abbiamo scoperto, quindi ad esempio in questo caso proviamo a modificare l'applicazione per rimuovere la demo. C'è una JC, dove si salta se c'è un carry, per cui cerchiamo di modificare il programma in modo da sostituire al jump una no-op: Ghidra permette di modificare le istruzioni, facendo la patch. È sconsigliato usarla come strada iniziale, perché Ghidra lavora su un DB dopo aver ricostruito cosa fa. Quindi bisogna fare la export del programma, ma questa operazione non è scontata, perché il programma è stato analizzato per costruire un DB, e fare le operazioni inversa non è scontato perché non si ottiene un file eseguibile identico a quello di partenza.
La patch del programma è un argomento avanzato, inizialmente si può fare direttamente sull'eseguibile, cercando di capire nel file sul disco dove è posizionata l'istruzione:

- andando col mouse sull'indirizzo, si mette fra le ultime cose l'offset nell'eseguibile sul disco;
- prendiamo l'exe e ne facciamo una copia, perché se facciamo un errore è un problema;
- ci sono diversi programmi per fare modifiche ai file binari, tra cui il programma commerciale WinEx;
- su Linux, c'è **okteta**, arriviamo agli stessi byte (72 18) e mettiamo delle no-op, ovvero due 90 90.

proviamo a lanciare il programma, sperando di vedere se la patch è andata. Ma in realtà non funziona, per cui dobbiamo cercare tutti i possibili riferimenti alle variabili o alle funzioni che gestiscono il timeout. Probabilmente l'approccio dell'aumentare il timeout può funzionare, ma dipende dal programma cosa funziona e cosa no.
Quindi, occorre cercare tutti i riferimenti al demo_length. Troviamo un altra CMP che verifica se il timeout è scaduto, quindi va patchato anche quello.
Ricordiamoci anche che AppDS può essere ottenuta in GetWindowLong, quindi magari lì abbiamo dei riferimenti alla demo_length che non riusciamo a vedere direttamente, per cui dobbiamo cercare i riferimenti anche alla GetWindowLong. Sono nella WinProc, quindi verifichiamo che non ci siano utilizzi della AppDS che usi l'offset della demo_length:

Chapter 6

Lezione 10

6.1 Rimozione finale della demo

Cerchiamo di vedere tutte le occorrenze della struttura, possiamo rapidamente cercare i campi coinvolti con la chiusura anticipata: troviamo un'occorrenza nella funzione che chiude Windows, questa volta la funzione è di 6 byte, ne prendiamo sempre l'offset passando il puntatore del mouse sulla finestra ed andiamo a salvare quanti byte annullare e dove.
Proseguiamo con la ricerca, fino a che non troviamo tutto ed a questo punto possiamo patchare il binario, agli offset indicati ci aspettiamo di trovare gli stessi byte individuati in Ghidra e sostituiamo con tanti 0x90 quanti necessari. Sappiamo già, dopo che il timer scade, l'applicazione continua a spegnersi: abbiamo visto solo i riferimenti ad AppDS, ma il dato viene anche salvato con la **GetWindowLong**, quindi occorre anche in questo caso cercare tutti in punti in cui si usa la procedura e vedere cosa si fa del valore. Vediamo che il valore preso viene messo nella variabile automatica l_appds. Troviamo il secondo confronto, nella procedura della finestra che arriva dalla gestione del comando paint: ogni volta che Windows chiede di ridisegnare la finestra, viene confrontato il valore del timer per verificare se non vada chiusa. Occorre perciò patchare questo ultimo punto. Il salto occupa 7 byte, quindi di nuovo andiamo a patchare il binario ed abbiamo completato l'operazione di rimozione della demo.
Abbiamo visto un'analisi statica di un eseguibile Windows per mezzo di un disassembler: il lavoro è lungo e ci vuole tempo per analizzare un codice "banale" e non troppo grande. Quindi analizzare del malware composto da codice più grande può essere molto lungo in base alla categoria a cui appartiene, per questo bisogna cercare delle strategie, possono esserci delle scorciatoie che permettono di arrivare più velocemente all'obiettivo cercato

6.2 Analisi dinamica vs analisi statica

Occorre iniziare a distinguere il concetto di analisi statica da quello di **analisi dinamica**: questi sono i due grandi blocchi con cui ci si confronta quando si fa reversing. L'analisi statica, a propria volta può essere vista come

- analisi statica di base
- analisi statica avanzata, quindi disassemblaggio dei programmi

la stessa divisione vale per l'analisi dinamica

- analisi dinamica di base
- analisi dinamica avanzata

Per ora, abbiamo visto solo analisi statica avanzata, cerchiamo di scoprire anche le altre. Abbiamo 3 approcci diversi:

- white box: "apriamo la scatola", per vederne il contenuto. Vi sono sia analisi statica di base che analisi statica avanzata;
- black box: in antitesi col primo, che corrisponde all'analisi dinamica di base. Consideriamo l'oggetto in analisi come qualcosa che possiamo eseguire ma in cui non possiamo entrare;
- gray box: corrisponde all'uso del debugger, quindi l'analisi dinamica avanzata.

6.2.1 Analisi statica di base

Questo tipo di analisi statica prevede l'analisi di un file binario senza l'utilizzo de disassembler. Il programma eseguibile è fatto di tante sezioni

- header
- text, ovvero il codice
- data
- rdata (read only data), come stringhe etc...
- rsrc, risorse nei programmi Windows. Sono dati strutturati fatti apposta per la GUI dell'applicazione

l'analisi statica di base analizza l'eseguibile in tutto ciò che contiene il programma senza fare il disassemblaggio del programma. Generalmente è più veloce dell'analisi statica avanzata e quindi generalmente è la prima cosa che viene fatta, potremmo trovare risposte a delle domande direttamente da qui

Tool per analisi statica di base

Vediamo alcuni fra i tool per fare analisi statica di base. Il SO target è Windows, ma alcuni tool possono essere anche usati in Linux, ci sono diverse classi di tool

Hashing: la prima famiglia di tool che possiamo usare è quella che serve per fare **l'hashing:** abbiamo un eseguibile e cerchiamo di capire se è già stato esaminato, o da me o da qualcuno, e ne abbiamo già informazioni. Il nome può cambiare, ma il contenuto no e quindi un modo semplice è calcolare l'hash del file con qualunque funzione di hash. Sull'hash dei programmi sono basati molti strumenti di ricognizione del malware: siti come VirusTotal collezionano motori anti-virus e di malware, se sottoponiamo il file questo ne registra l'hash in modo da poterlo riconoscere e dice se il programma è un malware o no.

Ricerca di stringhe nell'eseguibile Ci sono tool sia per Windows che per Linux, tr cui strings: questo fornisce tutte le sequenze di caratteri che possono essere ricondotte ad una stringa, per default cerca tutte le stringhe ASCII più lunghe di almeno 4 caratteri. È possibile passare delle opzioni per ad esempio cercare stringhe di meno di 4 caratteri o anche in formato Unicode. Funzionalità simili le ha strings per Windows, scaricabile a parte e lanciabile da cmd. Come mai è importante cercare le stringhe: spesso il tipo il stringa trovata suggerisce cosa fa il programma. Se ad esempio sono legate a problemi di connessione alla rete, sappiamo già che ci saranno delle connessioni di rete, uguale se troviamo errori di apertura di file etc...
Proprio per questo, chi fa malware come prima nasconde delle stringhe, quindi i programmi per la ricerca non danno informazioni utili in quanto queste vengono codificate per eludere tale ricerca. Inoltre, c'è un altro motivo per cui se si fa strings sul programma non si vede nulla

- l'eseguibile è stato compattato
- l'eseguibile è stato offuscato
- entrambe

Un **packer** prende le funzioni dell'eseguibile e le comprime, quindi l'eseguibile non è più fatto dal codice eseguibile ma da una versione compressa.
I packer si usano principalmente per ridurre spazio dell'eseguibile, ma anche per scoraggiare l'analisi statica del codice in quanto quando si apre il file in approccio white box si trova solo il codice di de-compressione. Quindi, il primo modo è disfare la compressione del packer, inoltre questi possono essere abbinati a degli **offuscatori**, che non solo comprimono ma scoraggiano anche chi cerca di ricostruire il codice. Ci sono vari offuscatori, per alcuni ci sono degli strumenti che de-offuscano, ma in generale analizzare il codice offuscato è una sfida, perché prima ci sono centinaia di ore di lavoro per de-offuscare → È SEMPRE UNA CAZZO DI GUERRA.

C'è una classe di tools per analisi statica che guarda solo l'header, che è molto ricco di informazioni sul programma

PEview Uno dei tool più vecchi è **PEview**: è il PE/COFF file viewer, aprendo un file ci da informazioni su:

- testate del file, quindi ad esempio IMAGE_DOS_HEADER etc... Le cose sono interessanti sulle testate IMAGE_NT_HEADERS,che contiene a sua volta altre testate. Ogni segmento del file ha una propria testata, che riassume delle informazioni relative ai singoli segmenti;
- all'interno dei singoli segmenti, ci sono degli indirizzi virtuali relativi al singolo segmento, che quindi vanno poi trasformati in indirizzi virtuali veri e propri quando il programma va in esecuzione
- ci sono poi i campi "Size of Raw Data" ed RVA, che sono due quantità pressoché uguali quando il programma non è compattato/offuscato, altrimenti lo spazio dei raw data occuperà circa la metà dello spazio finale;
- sezione iData, delle così dette import ovvero informazioni che servono per correlare l'eseguibile con le DLL. Ci sono quindi informazioni su quali DLL si usano e quali funzioni delle DLL si usano. Le informazioni su quello che fa il programma si possono dedurre

anche dagli headers, non sappiamo quando e se la funzione sarà eseguita ma è già una informazione

PE-bear Quando si cerca di rilevare il contenuto del codice offuscato, occorre modificare il contenuto delle testate e programmi come PEview non aiutano. Un programma poco noto ma utile è **PE-bear**, che permette di arrivare ai singoli byte e posizioni nel file di ogni elemento nel file, cosa essenziale se occorre modificare a mano gli headers del file. La correlazione fra posizione ed informazione non è banale, in quanto PEview da informazioni in termini di RVA, che però non è un vero offset nel file, ma lo è rispetto ad una sezione e quindi non so capire quale byte nel file contiene quella informazione. A differenza, PE-bear da questa informazione e permette anche di modificare a mano i valori.

LordPE Riassume brevemente le principali informazioni sul file eseguibile, quindi ad esempio dove è l'entry point, quante sezioni ci sono etc...
Anche questo è utile per vedere rapidamente le cose

PE-studio Programma più sofisticato, che sembra fatto apposta per analisi del malware: anche qui si carica il file e calcolare automaticamente le firme del file ed in generale da informazioni che quando si fa analisi del malware sono importanti

- data di compilazione del file;
- e così via

L'uso però è fondamentalmente simile a quello degli altri programmi: spesso quando si fa l'analisi per avere una prima idea è più rapido usare questi tool che aprire il programma con Ghidra per una prima analisi

Resource Hacker Serve a visualizzare le risorse in un file: le risorse sono di molteplici tipi

- bitmap;
- cursori;
- finestre di dialogo, che Resource Hacker apre e quindi già da questo possiamo capire cosa fa il programma. Siamo ancora in analisi statica, quindi non stiamo eseguendo il programma;
- risorse di stringhe, in modo da rendere nazionalizzabile il programma. Spesso molti tool che producono applicazioni mettono le stringhe come risorse
- manifest, ovvero una descrizione dell'applicazione
- risorse HTML

possiamo quindi vedere che risorse ci sono e di che tipo

PE Explorer È molto buono, ma ha come difetto di essere commerciale. Non da comunque cose in più rispetto agli altri

PEiD Fa parte dell'analisi dinamica del codice, quindi se uso dei plugin del programma si rischia grosso se eseguiamo il programma, occorre un ambiente controllato. Quando PEiD ha un programma offuscato, usa dei plugin che cercano il vero entry point e non quello dalla testata. Un'altra cosa molto utilizzata è un unpacker generico ed ha un **Krypto Analyzer** che analizza il codice cercando le costanti matematiche associate alle funzioni crittografiche: se il programma usa AES, questo ha cablato delle costanti nel codice e Krypto Analyzer le cerca nel programma e trova le istruzioni macchina che usano queste costanti, potendo così trovare le funzioni crittografiche nel codice.

6.2.2 Analisi dinamica di base

Nell'analisi statica viene preso l'eseguibile e trattato come qualcosa di immutabile, ovvero un artefatto e non si aspetta di eseguirlo.
L'analisi dinamica cerca di eseguire il programma per cercare di capire cosa fa, quindi sono richiesti dei tool differenti da quelli dell'analisi dinamica, alcuni di questi permettono di chiedersi

- quali sono le API, o chiamate di libreria, che l'eseguibile fa mentre è in esecuzione;
- quali sono i file aperti/chiusi;
- quali sono le chiavi nel registry key che il programma opera;
- quali chiamate alla rete esegue.

usiamo dei tool che monitorano tutti questi aspetti. È analisi dinamica di base perché non si entra nel codice del programma, se ne osserva solo il comportamento: facendo l'operazione sul malware bisogna stare molto attenti. Siccome l'analisi dinamica è molto più efficace nel dare informazioni rispetto all'analisi statica in termini di costi e tempi, chi scrive il malware cerca di proteggersi anche contro questo tipo di analisi.

Chapter 7

Lezione 11

7.1 Tool per l'analisi dinamica di base

Spesso, riusciamo ad ottenere informazioni sugli eseguibili senza aprirli, ma solo guardando l'interazione col SO. I tool per l'analisi di base dinamica sono vari

Process Explorer fa ciò che in Linux fa il programma ps, ovvero mostra una lista di tutti i processi attivi nel sistema, in ordine gerarchico.
Fa più di ps, in quanto può mostrare per ciascun processo una serie di proprietà sia legate all'eseguibile sul disco, sia legato al processo in memoria: i due mondi sono paralleli, ma non sono mai equivalenti in quanto il processo nasce dal file eseguibile, ma questo è solo una descrizione di come costruire il programma. Possiamo avere delle discrepanze nel codice eseguibile perché magari questo è offuscato, quindi i due mondi non coincidono. È possibile fare azioni sul processo, tra cui ad esempio poter creare un dump ovvero creare un immagine della memoria usata dal processo ed è interessante perché questo non coincide col codice del file eseguibile offuscato, quindi si potrebbe trovare il reale codice sorgente. Il codice offuscato è cifrato, compresso etc che non può essere passato direttamente al disassemblatore. C'è anche del codice in chiaro che instanzia il processo, si alloca una sezione testo, cioè codice, magari più grande di quella indicata dall'eseguibile. Questo perché il codice copierà i byte dall'eseguibile alla memoria ma magari espandendoli. Una volta finito questo, il codice fa il jump nel vero entry point, non in quello del codice offuscato. Lo schema è comunque semplice, se guardiamo dentro Process Explorer un dump della memoria, questo comprenderà il codice in chiaro, cosa che non avrei avuto aprendo il codice col deassemblatore.
Tramite la verifica della signature dell'immagine, possiamo verificare che la firma apportata da Windows sia verificata, in questo modo possiamo essere "sicuri" che l'eseguibile non sia malware. Questo però avviene sul file eseguibile, non sul processo in memoria: ci sono meccanismi in Windows che permettono di identificare un eseguibile come "buono", ma ci sono delle tecniche per sostituire il contenuto del processo con un altro eseguibile e quindi la facciata è la stessa, ma il processo che esegue no. Il contenuto dell'explorer ha tante sezioni, quella "stringhe" cerca sia quelle nelle sezioni del file eseguibile che quelle in memoria. Se le stringhe nell'eseguibile e quelle in memoria cambiano radicalmente vuol dire che il processo in memoria esegue del codice diverso da quello corrispondente all'eseguibile di partenza.

Process Monitor Monitor di sistema, ovvero registra tutto ciò che accade nel sistema. Accumula tutti gli eventi, c'è un modo però per fermare la cattura degli eventi quando sappiamo di non essere più interessati a raccoglierli, facciamo ripartire e poi cerchiamo fra quelli accumulati quelli che realmente interessano. La chiave per il filtraggio è dato proprio dai filtri che permettono, se la traccia di ciò che accade nel sistema è completa, di vedere un sotto-insieme degli eventi. Tutte le API che il processo utilizza vengono riportate nel monitor, ci sono cose non catturate in quanto il tool lavora a livello applicativo e quindi non cattura alcuni eventi legati alla gestione dei device driver di Windows, può essere rilevante perché alcuni malware si appoggiano ai device driver.

RegShot Nei sistemi Unix, si lavora con configurazioni memorizzate in file di testo editabili. Questa filosofia è stata stravolta da Windows, che memorizza le configurazioni in file dal formato non testuale, quindi in file binari, e in molteplici file del sistema. Ha poi dato un unico sistema per interfacciarsi coi file, che sono le chiavi di sistema: l'editor del registro di sistema fa vedere le migliaia di chiavi ed il loro valore, così come le può far modificare. Questo chiavi sono modificate in continuazione, fa parte della gestione standard dei processi di un SO, quindi diventa problematico capire come vi si interagisce ma se un malware vuole nascondere informazioni lo fa proprio nel registro di sistema. Con RegShot cerchiamo di capire come vengono modificate le chiavi del registro di sistema: dice come il registro di sistema è cambiato da un certo istante ad un altro. Ci sono due momenti:

- il primo snapshot va presa prima di eseguire il programma. Si può fare shot oppure shot and save
- il programma scandisce tutte le chiavi ed i valori associati

RegShot produce un report che dice

- quali sono i nuovi valori aggiunti;
- quali valori sono stati modificati.

Alcune delle chiavi sono standardizzate, ci sono dei nomi che hanno dei nomi abbastanza evocativi

Wireshark Sniffer dei pacchetti di rete, accumula tutto ciò che il SO fa vedere sull'interfaccia di rete con cui si può ricostruire il traffico di rete

apateDNS bisogna cercare di convincere il malware a comportarsi in modo da farci capire cosa fa. Una cosa che il tool fa è cercare di far connettere il malware ad un certo indirizzo DNS, invece di averlo cablato nel codice. ApateDNS fa due grandi cose

- sostituisce il servizio DNS di Windows;
-

Tipicamente viene usato il connessione con un altro tool, generalmente usato in Linux: si mette in piedi un infrastruttura per fare analisi del malware, mettendo su più macchine virtuali ed una di queste macchine può contenere il tool inetsim

inetsim consente di realizzare decine di servizi web fake. Tramite inetsim, possiamo ricreare un ambiente in cui simuliamo cosa accadrebbe se il malware fosse in esecuzione sul server reale, ad esempio configurando il DNS per far rispondere con un certo indirizzo ad apateDNS, così come anche http etc...

7.2 Utilizzo dei tool su un malware

La prima cosa da fare, è prendere uno snapshot della VM: quando ci si dispone ad eseguire malware, vogliamo salvare lo stato della macchina.
Nella sicurezza di Windows, possiamo disabilitare una serie di protezioni. Prima di questo, dobbiamo far si che queste modifiche siano persistenti altrimenti al riavvio saranno reimpostate. Sotto la protezione dagli exploit, disattiviamo le impostazioni (tutto in pratica).
Il problema di questo è che analizziamo un programma malevolo, che fa operazioni a basso livello, inoltre viene pensato per un SO target e non è detto che funzioni sul nostro SO e magari le differenze non sono configurabili. Per uscirne, occorre cercare una versione di Windows vulnerabile a quel malware, quindi tipicamente occorre una collezione di VM di Windows per capire quale permette di far girare correttamente il malware.
Prima di lanciare l'eseguibile, occorre far partire il tool di monitoring visti fin ora, apriamo ad esempio Process Explorer e Process Monitor. Facciamo partire il programma e vediamo che tutti i servizi col nome del nuovo servizio, ovvero svchost, sono figli di services.exe. Notiamo che se proviamo a far continuare il processo è in stato suspended e non si riesce a farlo andare avanti: il malware non sta funzionando anche con tutte le modifiche effettuate a Windows, quindi può essere che non sia compatibile con la versione del SO oppure che ci siano delle caratteristiche che gli impediscono di sostituire svchost.exe con un altro programma.
Proviamo a fare analisi statica del programma tramite PE studio.
Possiamo spostarci su versioni di Windows più vecchie, ma qui non è detto che abbiamo tutti i tool che avevamo sull'altra VM. Aprendo il programma e vedendone le stringhe, troviamo i tasti della tastiera e vediamo un file di logging quindi intuiamo che sia un keylogger. Mettendo il pid del programma in Process Monitor

- troviamo una CreateFile;
- scritture sul file;
- il file viene creato sul desktop, quindi troviamo al suo interno quanto digitato.

Chapter 8

Lezione 12

8.1 Analisi dinamica avanzata

Abbiamo visto tool sia per analisi statica che dinamica di base. Cominciamo a parlare dell'analisi dinamica avanzata, quindi l'uso del debugger.
Un debugger nasce come strumento per analizzare un codice e determinare se sta funzionando come il programmatore si aspetta, quindi localizzare tutti i punti in cui il programma fa qualcosa di differente da quanto pensato dal programmatore. Esistono diverse tipologie di debugger

- source level: si concentra sul programma ad alto livello scritto dal programmatore. Il debugger interagisce direttamente con le singole istruzioni ad alto livello, quindi ha senso se il linguaggio usato è ad esempio interpretato come Java e Python. Alcuni ambienti di sviluppo offrono debugger di livello source level;
- assembler level: cercano di ricostruire il funzionamento del programma dalle istruzioni assembly. Ci focalizziamo su questi, in quanto quando si analizza il malware non c'è codice sorgente.

Un'altra distinzione chiave è fra

- debugger kernel mode
- debugger user mode

parliamo di SO, perché fondamentalmente ogni volta che analizziamo il programma vediamo come questo interagisce con SO. Ogni processo può lavorare in modalità privilegiata o no, tipicamente il codice scritto dall'utente gira in user mode, poiché il SO lo isola per farlo coesistere con gli altri programmi, quindi ogni volta che il programmatore deve debuggare la sua applicazione, usa un debugger del secondo tipo. Il primo tipo può fare debugging anche del codice di SO: perché siamo interessanti anche ai debugger kernel mode? Ci sono vari tipi di malware ed alcuni si inseriscono direttamente nel SO, ad esempio il **rootkit**, che è un programma che modifica il modo di funzionamento interno del SO ad esempio nascondendo un'intera classe di processi ai tool ed agli analisti. Se un malware riesce a modificare il SO a livello kernel ha vinto, perché ora nella macchina non ci di può più fidare di cosa dice il SO, poiché è possibile che le risposte vengano modificate.
Una terza distinzione fra i debugger è data da

- quelli che lavorano in modalità remota;

- quelli che lavorano in modalità locale

i debug in modalità locale lavorano su un SO dove esegue sia l'applicazione da controllare e sia il debugger. Molti possono lavorare in modalità remota, ovvero servono due macchine: su una gira il target, sulla seconda il debugger e serve che esista comunicazione fra le due macchine e può essere una socket a livello SO o anche un collegamento fisico come un cavo USB, una porta seriale etc...
Supponiamo di star sviluppando un programma per un sistema embedded che usa un certo processore ma sto usando un cross-compiler, ovvero che produce codice adatto al sistema embedded, se devo fare debugging il sistema embedded non può contenere il debugger quindi occorre una soluzione del primo tipo.
Si applica anche a tutti i casi in cui non si vuole contaminare l'ambiente di lavoro in un caso in cui il malware possa infettare l'ambiente target: il debugger deve eseguire il malware e può accadere che questo sfugga dalle gabbie che il debugger crea.
Un'ultima categorizzazione

- debugger basati su GUI
- debugger a riga di comando

8.1.1 Ghidra

Ghidra è l'ambiente di riferimento per il disassemblaggio, il debugger è un'aggiunta recente (versione 10 >) ma si possono dire alcune cose al riguardo

- è un meta-debugger, ovvoernon ha un motore di debugging interno ma fa interfacciare Ghidra con dei debugger a più basso livello. Idealmente, chi sviluppa Ghidra sceglie di fare questo perché di impara una sola interfaccia, quella di Ghidra, potendo così cambiare il motore di debugging sottostante. Il secondo motivo è che Ghidra riesce meglio ad integrare quello che si scopre a livello di disassembler e a livello di debugging, però questa associazione è ancora molto primitiva
- Il tool di disassembleggio è un'interfaccia verso un DB dove vengono raccolte tutte le informazioni sul disassemblaggio, l'approccio al debugging è uguale. Il DB di debugging è chiamato traccia in Ghidra, l'idea è che raccogliendo informazioni sul debugging stiamo raccogliendo informazioni su una esecuzione specifica del programma. Un valore aggiunto del debugger di Ghidra è di salvare le tracce e di eseguirle avanti ed indietro, quindi si può navigare la traccia.

Il debugger è però ancora parecchio primitivo e quindi si riescono a fare solo poche cose essenziali con i debugger a basso livello, quindi occorre interagire direttamente col debugger a basso livello. Ghidra lavora sia in modalità remota che in modalità locale, è assembly level. I motori di basso livello con cui si interfaccia sono

- GDB
- WinDBG

8.1.2 IdaPro

Altro debugger molto diffuso ma che non useremo. È un ottimo disassemblatore, ha anche un debugger integrato in maniera dignitosa: funziona abbastanza bene l'integrazione fra disassembler e debugger

- è assembly level;
- principalmente user mode, ma si può anche usare in kernel mode;
- basato su GUI;
- permette di fare debugging sia a livello remoto che locale.

IdaPro ha un debugger interno ma può usarne anche altri come WinDebug. È un debugger dignitoso, non lo usiamo perché se paga e perché ci sono alternative migliori

8.1.3 GDB

È assembly level, nasce come debugger user mode (i meccanismi per fare debugging user mode sono più che altro trucchetti) a differenza degli altri è orientato alla riga di comando

- 'b' indirizzo mette un breakpoint all'indirizzo
- si possono aggiungere diverse GUI, il vantaggio è che essendo a riga comando è flessibile

molti ambienti di sviluppo integrano GDB, il vantaggio di questo debugger è che ovunque abbiamo codice per il quale è supportata la toolchain gcc è possibile usarlo. È quindi sempre possibile usarlo ed a cui ci si affida quando non si può usare altro. Non è però un debugger sofisticato come quelli di Win, ma fa il suo lavoro. GDB funziona sia in remoto che in locale

8.1.4 WinDBG

È il debugger ufficiale Microsoft. Sono uscite diverse GUI per questo debugger, in realtà i debugger veri e propri sono command line, quindi i motori veri e propri ricevono comandi da riga di comando e ci sono due possibilità

- alcuni dei debugger a basso livello sono user mode. Questi possono lavorare sia locale che remoto
- altri sono kernel mode, il lavoro è solo remoto: se si debugga kernel mode, si sta facendo debug del vero e proprio SO. Quindi, come si fa a mettere breakpoint nel SO e a far girare allo stesso tempo il debugger? Quindi, la filosofia è farlo in modo remoto, tramite VM e macchina fisica.

WinDBG è anche usato per debuggare i device driver sviluppati per il SO, quindi anche per fare debugging del codice del SO, i programmi a basso livello sono differenti ma le interfacce grafiche sono le stesse.
WinDBG è gratis ma non si scarica come prodotto a se stante, bensì con tanta altra roba dentro l'SDK di Windows.
C'è comunque la necessità di dover dare comandi a basso livello oltre alla GUI

8.1.5 SoftIce

Debugger vecchio, è assembler level e kernel mode ma locale: fa esattamente ciò che era impossibile fare per WinGBD, ovvero debuggare il SO all'interno di se stesso. Riusciva in questo compito, quindi era molto usato ma poi è scomparso: l'ultimo aggiornamento è del 2007 ma non è stato più aggiornato, quindi è fortemente legato al SO da debuggare, quindi ormai non si usa più.

8.2 Il debugger che useremo: OllyDbg

OllyDgb: è il debugger più ppopolare ed usato per fare debugging delle applicazioni Windows a 32 bit. Funziona solo per codice a 32 bit, quindi per malware a 64 bit occorre tornare ai debugger che supportano il 64 bit che sono WinDBG, GDB ed IdaPro.
È assembly level, user mode: non ci sono eccezioni per usarlo kernel mode, lavora solo in locale e basato su GUI. In termini di funzionalità è il miglior debugger in circolazione, ma ha questi limiti: si è tentato di superare i limiti, c'è la versione 2 che è un prodotto differente riscritto da 0. La versione 2 avrebbe qualche feature migliore della versione 1, ma chi fa analisi di malware o RCE non si fida della v2, viene ancora considerato una beta.
C'è stato un altro fork dalla versione 1, da cui è uscito fuori **ImmDbg**, che è una versione di un'azienda commerciale: è possibile estendere le funzionalità del debugger usando Python, tutti i debugger visti sono estendibili con script, ma il linguaggio è solitamente specifico del debugger.

8.2.1 Azioni nel debugger

Una delle prime operazioni da fare nel debugger è l'esecuzione step by step: quando si carica il processo in memoria, il debugger parte in modalità "paused". Il programma interattivo di solito aspetta l'interazione con l'utente, quindi bloccare l'esecuzione mi porta tipicamente a DLL che aspettano l'input utente. Quindi, la prima cosa da fare è il single stepping:

- f7, ogni volta che viene premuto, esegue una singola istruzione. La finestra sulla destra elenca il contenuto di tutti i registri del processore. Vengono sempre modificati i registri toccati dall'istruzione
- f8 è lo "step over".

f7 ed f8 sono praticamente equivalenti tranne che se si incontra una CALL: f8, ovvero "step over", la considera come singola istruzione e quindi esegue tutto come se fosse una singola istruzione e riblocca l'esecuzione all'istruzione seguente alla call; con f7 si entra dentro la call, è "step into". L'esecuzione step into è realizzato sui processori x86 con un flag particolare di registro di tipo "trap flag", ovvero con f7 il debugger imposta il trap flag nel registro di processore e quindi quando il codice verrà eseguito, automaticamente verrà alzata un'istruzione di debug: viene quindi gestito l'evento sincrono dell'eccezione (come ad esempio se ci fosse page fault, divisone per 0 etc...), quando si verifica l'eccezione di debug se il programma è stato registrato con un debugger, ovvero con un processo che lo controlla, il SO informa il processo che controlla dell'eccezione ricevuta e quindi il debugger può intervenire ed eseguire una singola istruzione.
Lo step over funziona col sistema dei breakpoint

Altra funzionalità importante è "execute till return": sono in una funzione di cui non mi interessa il contenuto e voglio ritornare a dopo la CALL, un'altra è "execute till user code": il debugger si ferma non appena capisce che è tornato ad eseguire codice dell'applicazione.

Finestre

C'è una finestra che indica i moduli eseguibili dell'applicazione: indica tutte le DLL usate, alcune sono sempre usate perché realizzano le syscall di Windows, altre invece sono presenti perché il programmatore ha scelto di usarle.

- Con "M" si apre la memory map, dove vengono elencate tutte le zone di memoria del processo: è un blocco contiguo di indirizzi di memoria che corrisponde a text, codice dell'applicazione, risorse etc...
- "T": finestra dei thread, possono essere creati dal SO, OllyDbg analizza l'intero processo, quindi si può sospendere l'applicazione ma di fatto OllyDbg non riesce a far mandare avanti gli altri thread se ne viene sospeso uno.
- "W": finestra che indica tutta la gerarchia delle finestre.
  "H": gli handle che Windows usa per fare riferimento agli oggetti dell'applicazione.
- "I": patch, permette di fare patch sia al programma in memoria che al file sul disco
  "K": finestra della call stack, analizza lo stack di ogni funzione e mostra i parametri passati ad ogni funzione

Chapter 9

Lezione 13

9.1 Ancora su OllyDbg

Ci sono diversi tipi di breakpoint, che vanno capiti bene: i due tipi fondamentali sono

- breakpoint di tipo esecuzione, ovvero che scattano quando si prova ad eseguire una certa istruzione macchina memorizzata in un certo indirizzo
- breakpoint di accesso, ovvero il processore prova a leggere/scrivere il dato dentro una certa cella di memoria

chi scrive malware cercherà di ostacolare chi fa analisi, mettendo in piedi dei meccanismi per rilevare se sono stati inseriti dei breakpoint nell'eseguibile. Un'altra differenza nei breakpoint è quella fra

- di tipo software, ovvero gestiti dal debugger in maniera pesante
- di tipo hardware, sono in qualche modo implementatati dal processore "gratis" ovvero forniscono una metodologia di breakpoint a più alto livello

9.1.1 Breakpoint di tipo esecuzione software

Servono due meccanismi per l'implementazione

- interrompere l'esecuzione all'istruzione successiva a quella in cui mi trovo: faccio partire il programma è poi do f7 da ollydbg, in questo caso il breakpoint viene realizzato tramite il trap flag, quindi ogni volta che viene eseguita una istruzione macchina si genera un'eccezione di debug, in questo caso asserita perché ogni istruzione macchina setta il flag TF;
- uso delle istruzioni macchina dette INT3, tali istruzioni se eseguite comportano da parte del processore l'asserzione della debug exception. In ollydbg si può fare con tasto destro oppure f2, il debugger prende il contenuto in memoria corrispondente al primo byte dell'istruzione macchina, lo salva ed al suo posto mette il codice operativo della INT3;
- memory o page protection: in ollydbg si può aprire la memory map che contiene le regioni di memoria in cui è organizzato il processo. In ciascuna di queste si possono settare dei breakpoint sull'accesso. Non appena l'eseguibile accede alla sezione di memoria del

programma (dove è settato il breakpoint), il debugger interviene: prende tutte le pagine in memoria associato al segmento e ne modifica il flag di accesso in modo che si alzi una eccezione di tipo page fault

- breakpoint di tipo esecuzione hardware, in quanto il processore ha dei registri speciali che possono essere usati per programmare alcuni breakpoint hardware, ovvero gestiti direttamente dal processore. All'interno di ollydbg può essere anche disabilitato, inoltre in "hardware breakpoint" definiamo l'evento che porta al breakpoint. Sono molto flessibili ed efficienti, ma hanno come grosso limite di essere solo 4, sono molto più utili per gli accessi. Se dobbiamo mettere un breakpoint software di accesso, il debugger deve cambiare tutta la pagina per poter catturare l'eccezione e quindi l'overhead è più grande, con quello hardware è più semplice.

I breakpoint sono persistenti, quindi anche se si va avanti rimane nel debugger. Le contromisure che si possono mettere in atto sono

- vedere se il trap flag è settato, per il malware è un chiaro indice che qualcosa non va;
- molti malware hanno routine che prendono tutto il proprio codice in RAM e cercano le istruzioni INT3 e si rendono conto che c'è un debugger che cerca di eseguire

int3 si può rappresentare sia con un codice da 1 byte che con uno da 2 byte

9.1.2 Breakpoint in accesso

Qui si usano sopratutto breakpoint hardware, ci sono delle contromisure perché anche il malware può controllare i registri per vedere se i valori contenuti sono stati inseriti dal malware stesso oppure no. Ci sono anche breakpoint software, sono basati sulla protezione delle pagine e quindi abbastanza lenti.

9.1.3 Breakpoint condizionali

Sono breakpoint come tutti gli altri, in OllyDbg solo quelli software possono essere condizionali (scelta progettuale), ma sono comunque attivi: quando si arriva ad una condizione o si esegue una istruzione macchina il debugger interviene e controlla una condizione, se falsa si continua col flusso di esecuzione, altrimenti si ridà il controllo all'utente.
Si può mettere sempre col tasto destro, va messa una condizione che deve essere vera, si usa il solito meccanismo della [] per l'indirezione.

9.1.4 Gestione delle eccezioni

L'eccezione è tale per cui nel flusso di esecuzione c'è una istruzione che fa si che il flusso di esecuzione venga dirottato o ad una procedura nell'applicazione oppure da un handler predefinito dal SO. Il debugger permette una 3° possibilità in quanto al verificarsi dell'eccezione il primo ad esserne informato è proprio il debug, nella fase detta **first_chance** handling, ovvero c'è una prima possibilità di intervenire sulla eccezione:

- può scegliere di buttare via l'eccezione

- può scegliere di non gestirla e restituirla al programma sotto debugging, come se il debugger non esistesse
- può mettere un breakpoint in caso di eccezione

c'è poi una **second_chance**, può accadere che il debugger dica di passare l'eccezione al gestore definito nell'applicativo, ma se mentre si gestisce l'eccezione ne capita un'altra, spesso dello stesso tipo, questo è problematico e quindi spesso non si gestisce la second_chance perché queste seconde eccezioni spesso sono legate a dei bug legati al programma originale. Nelle opzioni di debugging possiamo indicare quali sono le eccezioni da non gestire e da passare direttamente al programma, in quanto magari sono dovute a problemi del programma stesso; si possono anche ignorare eccezioni di un certo range.
Le eccezioni sono largamente usate dal malware per identificare la presenza di debugger

9.1.5 Trovare il main loop usando il debugger

Al caricamento del programma, siamo nell'entrypoint del programma, quello che possiamo fare è provare ad eseguire passo passo il programma come step over, ovvero considerare la CALL come unica istruzione: se alla fine della CALL c'è il main loop non se ne uscirà più finché non termina il programma, perché appunto è un loop. Il tasto f4 permette di fare la run fino alla riga selezionata, viene fatto con un breakpoint non persistente. Metto un primo breakpoint sulla CALL, ogni volta che vado avanti e perdo il controllo metto un breakpoint sulla CALL successiva, se troviamo dei cicli andiamo avanti con f4. Troviamo il MessageLoop, probabilmente abbiamo trovato la WinMain, anche OllyDbg ha la capacità di associare delle etichette agli indirizzi, come disassemblatore Ghidra è migliore quindi conviene lavorare in parallelo.

9.2 Analisi di un file di laboratorio del libro di testo

Analisi del programma "lab9".
Si parte sempre facendo uno snapshot pulito, va anche disabilitato il riconoscimento del malware. Una delle prime cose utili da fare può essere aprirlo con PEiD, aprendolo in Ghidra proviamo a cercare le stringhe

- vediamo delle stringhe che fanno riferimento al protocollo HTTP
- stringhe download, upload, magari carica/scarica contenuti
- cmd.exe, il programma fa qualcosa con la shell dei comandi di Windows
- %SYSTEMROOT%: in Windows, il meccanismo di accesso alle variabili di ambiente usa i "%", mettendosi in %SYSTEMROOT%\system32 si mette nella cartella che contiene i programmi per eseguire il codice a 32 bit.

Appendice

Comandi utili per Ghidra

- con il tasto "G" si apre il box per saltare in un certo punto del file, dando un'etichetta o un indirizzo esadecimale;
- passando su un valore esadecimale, è possibile passarvi sopra per vedere il valore. Col tasto destro si può convertire il valore;
- è possibile richiedere un call graph, che rappresenta con ogni punto una funzione e rappresenta tutte le call che la funzione fa. Rispetto ad IDA; Ghidra riesce a mostrare le call solo per il livello successivo del grafo, poi bisognerebbe andare avanti a mano;
- symbol tree definisce tutti i simboli usati da Ghidra, come anche tutti i simboli che sono stati riconosciuti come funzioni;
- con "X" si trovano tutti i riferimenti ad una chiamata di funzione;
- ";" permette di inserire i commenti, in quanto in assembly si fa così;
- tasto dx sulla funzione → edit stack frame: permette di modificare le variabili sullo stack, ad esempio aggiungendone il tipo
- memory map: finestra che elenca tutti i "segmenti" di memoria su cui Ghidra ha costruito il reversing. Importante in quanto ogni tanto è necessario far capire a Ghidra come è fatto un certo segmento, da un quadro di insieme della struttura dell'eseguibile
- per aprire i riferimenti a variabili, tasto dx sul simbolo per andare sui riferimenti. vengono mostrati riferimenti al simbolo o all'indirizzo, vogliamo generalmente quelli all'indirizzo, c'è la scorciatoia con "X". Questa funzionalità si usa di continuo.
- menù di ricerca, si può cercare sia nelle istruzioni macchina che nella memoria. Fondamentalmente, quello sulla memoria è più generale, ci sono poi sotto-menù che fanno ricerche a grana più fine