

Malware Analysis

1. Analisi statica & dinamica - 2 novembre 2023

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

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

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

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

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

2. Analisi statica dinamica - 7 novembre 2023

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

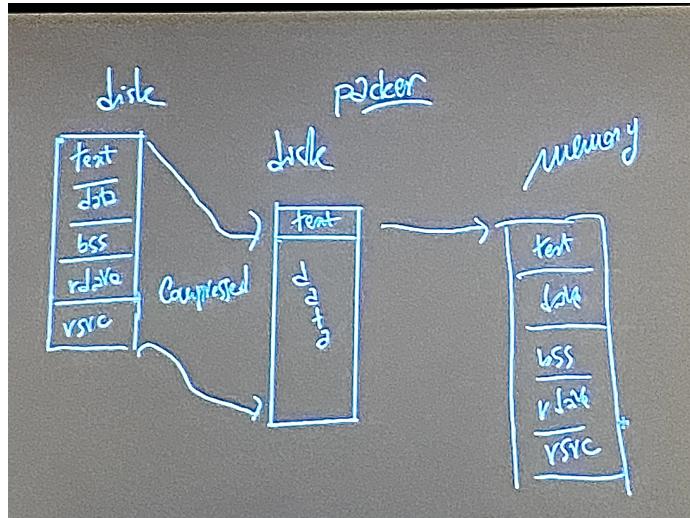
2.1. Process Explorer

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

Sappiamo che un file eseguibile è composto da *testo*, *data*, *bss*, *risorse*.

Per risparmiare sulle dimensioni dell'eseguibile, si usano i *packer*, che prendono queste info e le trasformano in un nuovo eseguibile, con una parte *testo* piccolissima, e il resto è tutto compresso, decomprimendolo poi in memoria. Poi fa jump a prima cella del codice originale. Quando viene caricato in memoria, ricrea la forma originale. Se confronto con la versione compressa, ovviamente avrei un

offuscamento, dobbiamo confrontare la versione non compressa nel disco (anche la versione compressa è



nel disco) e memoria.

2.2. Process Monitor

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

2.3. Regshot

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

2.4. Wireshark

Usato per flussi a livello di rete.

2.5. ApateDNS

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

2.6. ApiMonitor

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

2.7. Esempio di analisi w03-03

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

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

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

2.7.1. Come passo il malware su VM?

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

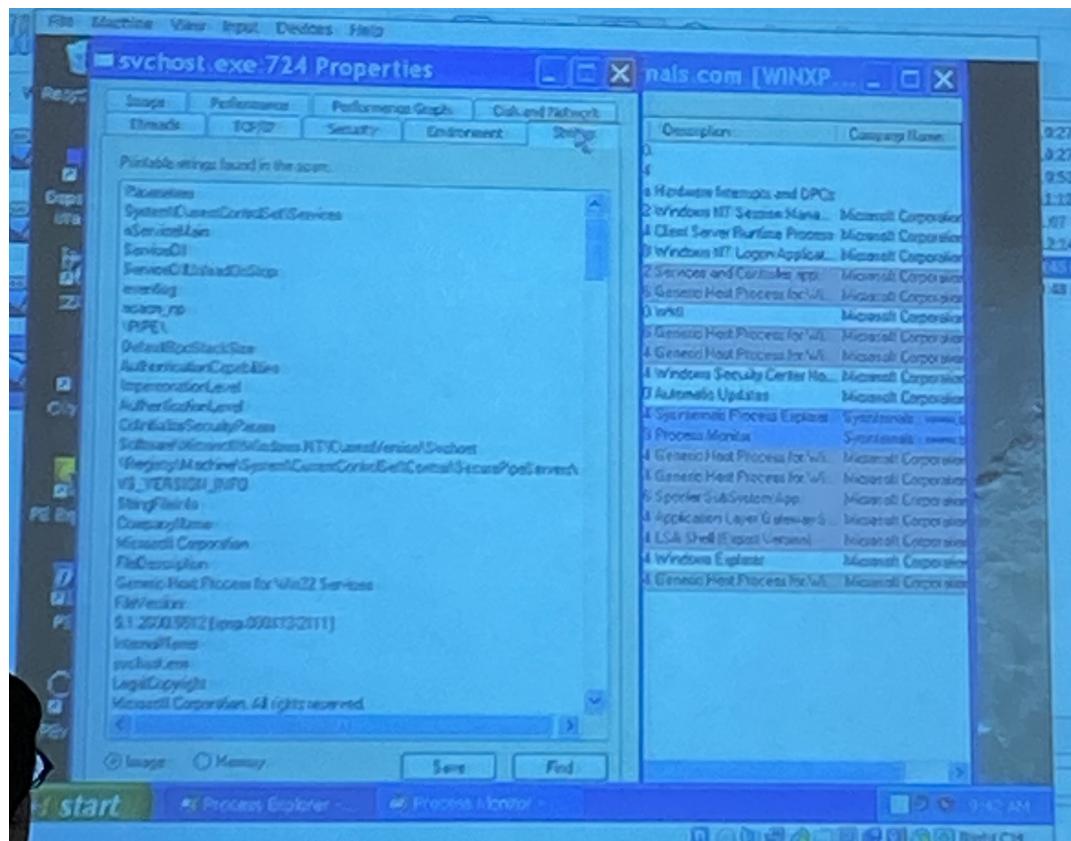
2.7.2. Inizio analisi

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

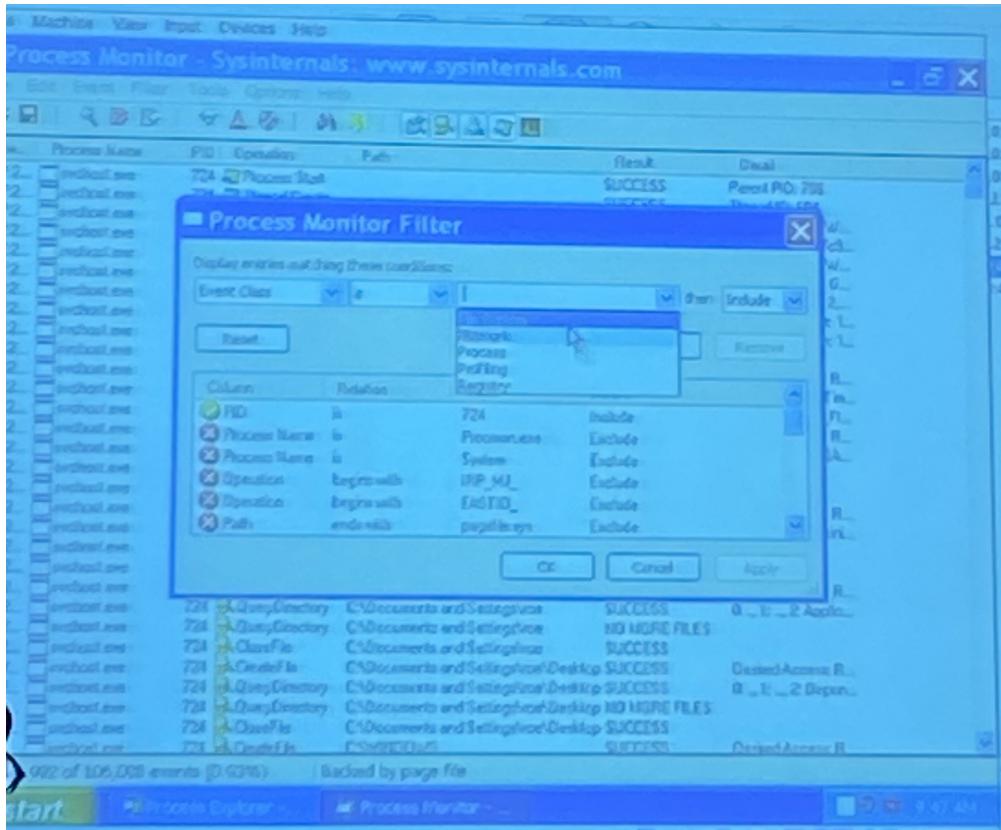
2.7.3. Parte 2, 9/11/23 - VirtualBox

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

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



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



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

3. Analisi ed intro al Debugger

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

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

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

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

Alcuni debugger lavorano in modalità *locale*, altri in *remota*, o ancora *mista*. Il locale gira sulla stessa macchina del processo esaminato, nella modalità remota c'è disaccoppiamento. Esempio in questo

secondo caso, malware *Android*, il debugger lo metto su un computer, è sicuramente più comodo.

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

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

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

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

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

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

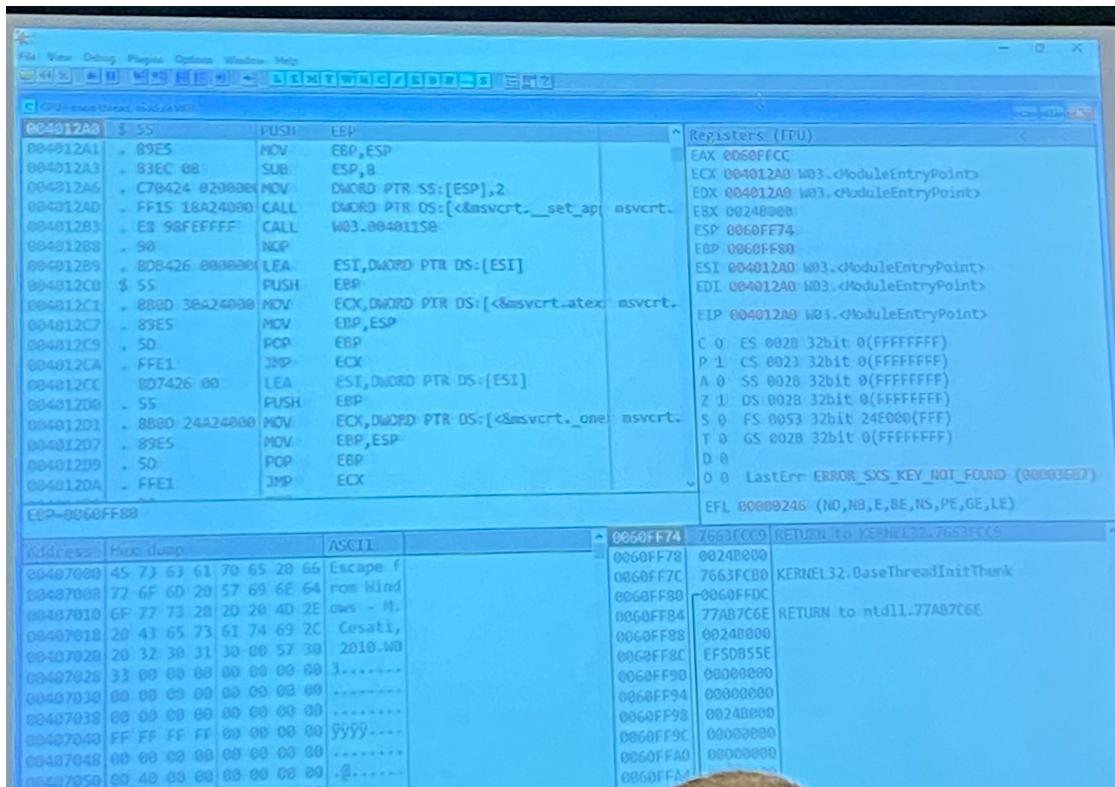
Oggi si preferisce usare **OllyDbg versione 1.10**, anche lui vecchio, ma comunque con molti plugin (esempio: auto identificazione di strutture dati note). E' solo *usermode 32 bit Intel*, di tipo *GUI*. Da lui deriva *OllyDbg 2*, interfaccia uguale, ma riscritto internamente. Lento, pochi plugin. Il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

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

3.0.4. Overview su Olly

- *In alto a sx*: disassemblato, parte da entry point, sintassi intel (destinazione a destra).
- *In alto a dx*: finestra registri, il loro contenuto, tutti! (follow in dump per seguire uno specifico registro). Interessante è *debug flag T*, bit che, quando viene generata eccezione, viene impostato ad 1 prima di eseguire, genera trap per il sistema operativo, che invece di segnalarla al processo (che sarebbe il naturale corso degli eventi) la segnala al debugger. Con il comando *step into F7*, decidiamo noi come proseguire in questi casi.
- *In basso a sx*: finestra sui dati, vedo quello a cui punta un registro, vedendo la sua evoluzione.

- *In basso a dx:* finestra sullo stack, che cresce per indirizzi decrescenti, su indirizzi superiori ci sono vecchie posizioni dello stack.



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