

INFORGE.NET

Cheat Engine

Il memory-scanner

by

your hacking experience



0.1 - Prefazione

Con questa guida impareremo a [memorizzare](#) e a [gestire](#) la memoria e non solo. Il testo tratta anche alcune basi del linguaggio [C](#) (riportando ad alcune pagine web esterne), spiega come funzionano i [processori](#) nei sistemi operativi *Windows* (in modo molto semplificato), come il [kernel](#) esegue le istruzioni di un programma, eccetera... Inoltre oltre alla modifica normale della memoria viene spiegato anche il [debugging](#) e altre tecniche che vi potranno essere utili nella creazione di [programmi](#). Insomma, con questa guida imparerete a lavorare bene con [la memoria](#).

In questa guida non vi fornisco [indirizzi](#), o [link](#) per i vostri giochi, vi insegno invece a trovarli. Quindi scordatevi di poter cercare qui i vostri address.

Durante la guida vengono usati i giochi di *Windows* e altre applicazioni scaricabili facilmente. Nel video-guida uso anche il gioco *Metin2*. Uso il client di babau, scaricabile facilmente cercando con *Google*, per i server privati, perché *Metin2 IT* è protetto da *HackShield*. Mi scuso per la scarsa qualità del video.

Il testo, diviso in unità e capitoli, inizia con le premesse di carattere [teorico](#) e [pratico](#), per poi passare alle basi del linguaggio [C](#). Seguono le unità per il [debugging](#), il [kernel](#) e il [sistema operativo](#). Alla fine viene ripreso tutto dall'inizio, spiegando le funzioni che ogni finestra di Cheat Engine fornisce.

Come prerequisiti non c'è niente di necessario. Ma avere buona conoscenza del funzionamento di un computer e del linguaggio [C](#), aiuta moltissimo.



Sono convinto che l'informatica abbia molto in comune con la fisica. Entrambe si occupano di come funziona il mondo a un livello abbastanza fondamentale. La differenza, naturalmente, è che mentre in fisica devi capire come è fatto il mondo, in informatica sei tu a crearlo. Dentro i confini del computer, sei tu il creatore. Controlli – almeno potenzialmente – tutto ciò che vi succede. Se sei abbastanza bravo, puoi essere un Dio. Su piccola scala.

Linus Torvald

Unità 1

Informatica e Matematica

- Sistemi di numerazione
- Rappresentazione dei numeri interi

1.1 – Sistemi di numerazione

Noi usiamo il sistema decimale (o base 10), ovvero usiamo 10 cifre per rappresentare qualsiasi numero:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

In informatica però vengono spesso usati sistemi di numerazione di base diversa. I più usati sono il sistema binario (o base 2) e quello esadecimale (o base 16). Più raramente viene usato il sistema ottale (o base 8).

Dobbiamo però prima imparare a scrivere un numero in notazione scientifica. Prendiamo ad esempio il numero 784, in notazione scientifica diventa:

$$784 = 4 \times 10^0 + 8 \times 10^1 + 7 \times 10^2$$

Come vedete un numero si rappresenta in notazione scientifica prendendo l'ultima cifra a partire da sinistra e moltiplicandola prima per 10^0 poi per 10^1 e così via... aumentando per ogni cifra l'esponente del 10.

Perché proprio il 10? Perché è in sistema di numerazione in base 10, se lavorassimo in base 2 allora si moltiplicherebbe per 2^n , dove n è la posizione della cifra da destra verso sinistra partendo da 0.

Quindi, generalizzando:

In un sistema di numerazione in base b , un numero si rappresenta in **notazione scientifica** sommando il prodotto di ogni cifra del numero per b^n , dove n è la posizione della cifra da destra verso sinistra dove la cifra meno significativa occupa la posizione zero.

La cifra meno significativa è sempre la cifra più a destra, è quella delle unità, quella che vale meno.

Il sistema binario usa invece solo due cifre, lo 0 e l'1, chiamate *(Binary Digit)*. Anche le cifre del sistema binario, come quelle del sistema decimale, assumono un valore, ovvero più sono "a sinistra" più il loro valore è alto. Vediamo un esempio:

$10110101_2 = 181_{10}$ (i numeri piccoli in basso indicano la base in cui viene scritto il numero)

Infatti: $1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = 1 + 0 + 4 + 0 + 16 + 32 + 0 + 128 = 181$.

Per convertire invece dalla base 10 alla base 2 si divide per 2 il numero dato e si scrive il resto (che può essere solo 0 o 1); il quoziente ottenuto viene a sua volta diviso per 2 ottenendo un nuovo resto; si prosegue fino a quando si ottiene come quoziente il numero 0; Alla fine si leggono i resti

. Esempio:

$$35_{10} = 100011_2$$

$$35 / 2 = 17 \text{ e resto } 1$$

$$17 / 2 = 8 \text{ e resto } 1$$

$$8 / 2 = 4 \text{ e resto } 0$$

$$4 / 2 = 2 \text{ e resto } 0$$

$$2 / 2 = 1 \text{ e resto } 0$$

$$1 / 2 = 0 \text{ e resto } 1$$

Leggendo i resti al contrario si ottiene appunto: 100011_2

Risparmio le operazioni in base 2 perché è sufficiente una calcolatrice. Per chi volesse studiare come si eseguono le quattro operazioni fondamentali in basi diverse da quella decimale (non cambia molto) può documentarsi su internet.

Il usa invece 8 cifre:

0, 1, 2, 3, 4, 5, 6, 7

I metodi di conversione sono gli stessi:

$$265_8 = 181_{10}$$

$$\text{Infatti: } 5 \times 8^0 + 6 \times 8^1 + 2 \times 8^2 = 5 + 48 + 128 = \quad .$$

Al contrario:

$$35_{10} = 43_8$$

$$35 / 8 = 4 \text{ e resto}$$

$$4 / 8 = 0 \text{ e resto}$$

Leggendo i resti al contrario si ottiene appunto: 43_8

Il è invece un po' più complesso, esso utilizza 16 cifre:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

La lettera A rappresenta l'11 decimale, la B il 12 decimale, e così via fino a 15.

I metodi di conversione sono sempre gli stessi, a questo punto penso di poterli omettere. Basta sostituire alle lettere A, B, C, D, E e F i corrispondenti numeri decimali prima di eseguire la conversione.

Risparmio le conversioni tra le basi 2, 8 e 16 perché basta dotarsi di una calcolatrice. Consiglio comunque di informarsi con internet.

1.2 – Rappresentazione dei numeri interi

In informatica esiste una scala per rappresentare i numeri binari, i :

1 byte = 8 bit

2 byte = 1 word = 16 bit

4 byte = 2 word = 1 double word = 32 bit

8 byte = 4 word = 2 double word = 1 quad word = 64 bit

È importante imparare a memoria questa tabella.

Supponiamo di usare una double word (16 bit), possiamo quindi rappresentare numeri interi positivi che vanno da 0 a $2^{15} - 1$, ovvero da 0 a 32767.

Il segno viene rappresentato dal bit più significativo (ovvero quello più a sinistra). 0 per il segno + e 1 per il segno -.

Per i numeri negativi si ricorre invece al criterio del :

Dato un numero espresso in forma binaria, per ottenere il complemento a 2 del numero dato, ad ogni cifra si sostituisce 0 dove c'è 1, 1 dove c'è 0 e alla fine si somma 1.

Esempio:

dato il numero: 10111001

si scambiano le cifre: 01000110

si somma 1: +1

complemento a 2: 01000111

Possiamo verificare che 10111001 corrisponde a 185 e 01000111 corrisponde a -185.

Se si riesegue il complemento a 2 si otterrà il numero di partenza.

Unità 2

Dietro il monitor del computer

- Struttura generale del computer
- I Bus
- La C.P.U.
- La memoria principale
- La memoria di massa
- Le unità di input e output
- Prima di procedere...

2.1 – Struttura generale del computer

In generale, un computer si compone delle seguenti parti fondamentali:

1 – C.P.U. ovvero, *Central Processing Unit*, Unità Centrale di Processo.

2 – Memoria principale.

3 – Memoria di massa.

4 – Unità di input (ingresso).

5 – Unità di output (uscita).

2.2 – I Bus

Abbiamo parlato delle componenti principali dell'hardware di un computer. È ovvio che queste componenti comunicano tra di loro e quindi dovranno essere collegate. Il trasporto delle informazioni viene effettuato dai bus del computer, ovvero dei semplici gruppi di fili. Ognuno di questi fili rappresenta un bit. Quando passa la corrente elettrica da questi cavi, il bit rappresentato da quel filo acquista valore 1, in caso contrario 0. È logico pensare che i bus sono composti da un alto numero di fili per rappresentare numeri molto grandi. Un bus a 32 bit (quindi con 32 fili) sarà in grado di rappresentare un numero composto da un massimo di 32 cifre binarie. In un computer esistono 3 tipi di bus:

- Bus dati (Data Bus)
- Bus degli indirizzi (Address Bus)
- Bus di controllo (Control Bus)

Il _____ è il bus che provvede allo scambio dei dati. Più grande è il bus, più veloce sarà lo scambio. Attualmente i computer hanno per la maggior parte bus dati di 32 bit, e si stanno diffondendo computer con bus dati a 64 bit.

Il bus dati viene gestito dalla CPU, che decide le letture e le scritture da effettuare in tutti gli altri componenti del computer.

Il _____ è invece quel bus che serve appunto alla CPU per richiamare le varie componenti. Ad esempio se la CPU "vuole" leggere un valore salvato in memoria, invierà alla memoria, tramite il bus degli indirizzi, il messaggio di farsi inviare ciò che desidera leggere, a questo punto la memoria utilizzerà il bus dati per inviare le informazioni richieste alla CPU. Il bus degli indirizzi parte dalla CPU con 20 fili (bit) e si divide in 2 parti composte sempre da 20 bit. Una parte va alla memoria principale, e l'altra parte si divide a sua volta in 2 parti (ognuna composta da 10 bit) e, una va ai dispositivi di input, e l'altra ai dispositivi di output. Rimane un dubbio: se la CPU deve richiamare un componente come appunto la memoria principale, come fa utilizzando solo lo 0 e l'1? Semplice: ogni componente possiede un nome binario, e la CPU può richiamarli utilizzando tale nome (ad esempio: 0000011010101110).

Il _____ è un po' più complesso e non lo analizzeremo.

2.3 – La C.P.U.

L' _____ è il cervello del computer e si compone in 2 parti:

- L'A.L.U. (*Arithmetic Logic Unit*), unità aritmetico logica.
- L'unità di controllo

La prima esegue le operazioni aritmetiche e logiche, mentre l'unità di controllo si occupa di attivare le componenti che fanno parte di tutto il computer.

Attorno alla CPU si trovano i _____, che possiamo vedere come piccole memorie, rapidissime nello scambio di informazioni, che si occupano di contenere piccole informazioni al fine di elaborare i dati e organizzare il loro smistamento. Esistono vari registri in un computer, e ognuno è specializzato a svolgere determinate operazioni.

2.4 – La memoria principale

La memoria va vista come una tabella, dove a destra si trovano gli **indirizzi** e a sinistra i **valori**.
L'associazione tra un indirizzo e un valore è chiamata **memoria principale**. Gli indirizzi sono un numero intero e i valori sono delle informazioni di 1 byte (8 bit). Esempio di un' **array**:

Indirizzo	Valore
1	00101101
2	01010011
3	10100010
4	00100110

Ogni riga è una locazione di memoria. Quando la RAM vuole leggere il valore dell'indirizzo 2, utilizzerà il bus degli indirizzi e quello di controllo (che non abbiamo visto) per farsi inviare l'informazione tramite il bus dati.

La memoria principale si divide in 2 parti:

- ROM (*Read Only Memory*, Memoria di sola lettura) dove sono contenute tutte le istruzioni che la C.P.U. deve leggere e interpretare per poter svolgere qualsiasi funzione.
- RAM (*Random Access Memory*, Memoria ad accesso casuale) dove la CPU legge e scrive tutti i dati e le istruzioni dei programmi. La RAM è chiamata anche "memoria volatile" perché si svuota quando viene spento il computer.

2.5 – Le memorie di massa

Le memorie di massa sono dispositivi che contengono dei dischi su cui è possibile scrivere e leggere. Tutti i computer possiedono internamente una memoria di massa, l'*hard disk* (o *disco-rigido*). Esso è solitamente molto ampio, e contiene tutto ciò che salviamo sul nostro computer. Anche i pen-drive, i CD-ROM e qualsiasi altro tipo di memoria esterna sono delle memorie di massa. A differenza della memoria RAM, le memorie di massa non sono “volatili”.

2.6 – Le unità di input e output

Il computer sarebbe completamente inutile se non interagisse con l'esterno. A cosa servirebbe infatti una macchina che non ci permette di fare niente?

Per questo esistono le unità di input (come la tastiera, il mouse, il microfono, la webcam, lo scanner, ecc.) e di output (come il monitor, la stampante, le casse, le cuffie, il proiettore, ecc.). Queste unità speciali permettono di inviare al computer delle informazioni (input) che verranno poi elaborate, e verrà poi (forse) restituito qualcosa (output).

Una macchina o anche un software che interagisce con l'esterno si definisce . Vice-
versa, una macchina o un software che non interagisce con l'esterno si definisce .

2.7 – Prima di procedere...

Prima di procedere con questa guida consiglio fortemente di leggere questa pagina:

http://quequero.org/Lezione_1_Assembly

Non procedete senza averla letta perché non capireste niente. Essa spiega più in dettaglio come funziona il computer e poi fornisce un po' di basi del linguaggio *assembly*. Se inoltre volete imparare davvero bene potete andare anche oltre la prima lezione, studiando prima tutto l'assembly. La prima lezione è però .

Chi proprio volesse il meglio, invece consiglio di studiarsi TUTTA la guida alla pagina:

<http://xoomer.virgilio.it/ramsoft/>

Qui partite dalla sezione "Assembly base (con MASM e TASM)" poi "Assembly Avanzato" e infine "Win32 Assembly". Se studiate questo non avrete poi la minima difficoltà a lavorare con Cheat Engine.

Unità 3

Memory-scanning

- Memory scanner: Cheat Engine
- Tipi di dato
- Tipi di ricerca
- Prima scansione
- Dynamic Movement Address
- Pointer scanning

3.1 – Memory scanner: Cheat Engine

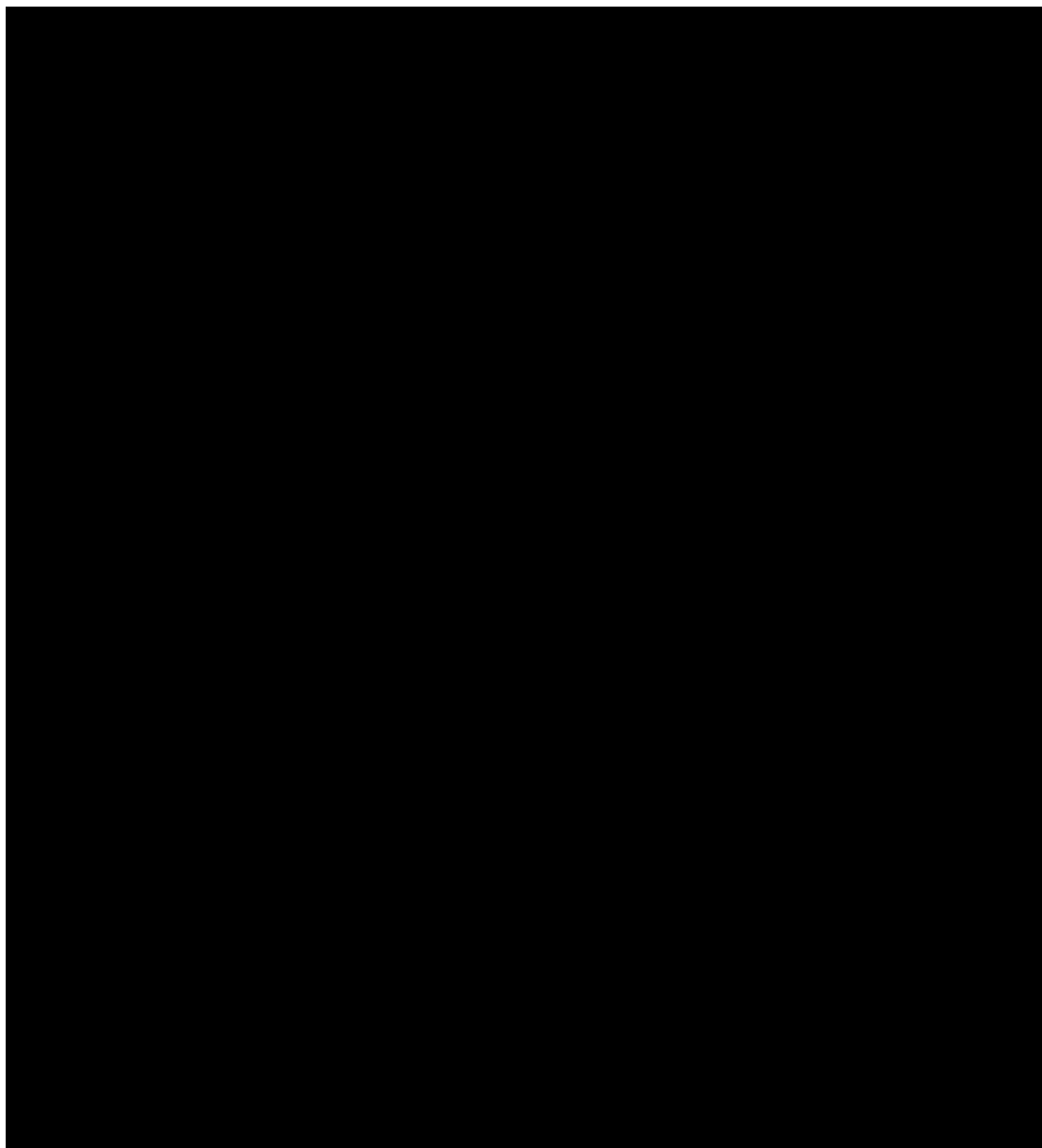
è probabilmente il più famoso, e forse anche il più potente, gratuito e scaricabile dal web. È possibile scaricarlo dal sito:

<http://www.cheatengine.org/>

Cheat Engine non è solo un memory-scanner, ma anche un editor di memoria, e un debugger, e molto altro ancora! Noi impareremo alcune di queste funzionalità.

3.2 – Tipi di dato

Abbiamo già visto i vari tipi di dato, ma adesso guardiamoli su CE (Cheat Engine). Apriamolo. La schermata che ci comparirà sarà come la seguente:



Ora aprite un processo qualsiasi (io ad esempio ho scelto il processo *Skype*). Per aprire un processo dovete fare click sul computerino lampeggiante in alto a sinistra, e nella schermata che compare, selezionare il processo che si desidera aprire e poi fare click su *Open*.

Nella parte alta a destra del programma c'è una grossa TextBox. Quella è la TextBox per la ricerca. Sotto vediamo ***Scan Type*** e ***Value Type***. *Value Type* indica il tipo di valore da trovare in memoria. Abbiamo già visto ai paragrafi 1.2 e 1.3 i vari tipi, e qui sono gli stessi.

3.3 – Tipi di ricerca

Abbiamo visto *Value Type*, adesso guardiamo ***Scan Type***. Questo imposta il metodo di ricerca. Inizialmente troviamo i seguenti:

- ☐ **Exact Value** = Valore esatto. Ricerca un preciso valore in memoria.
- ☐ **Greater Than** = Più grande di... Ricerca tutti i valori maggiori di un numero specificato nella TextBox.
- ☐ **Less Than** = Come sopra, ma cerca i valori minori.
- ☐ **Between** = Valore compreso. Ricerca tutti i valori compresi in un intervallo specificato.
- ☐ **Starts With** = Valore iniziale sconosciuto. Si utilizza quando non si conosce il valore da cercare.

Ora selezionate *Exact Value*, e mettete un numero qualsiasi nella TextBox. Poi cliccate su *First Scan*.

Dopo un po' a lato vi compariranno tutti gli address (indirizzi) trovati, con il relativo valore a fianco. Ma non è questo che voglio farvi vedere. Se ora vedete in Scan Type sono comparse più opzioni. Queste nuove opzioni servono a filtrare tutti gli address trovati fino ad ora. Infatti i valori degli address cambiano in continuazione, quindi sono facilmente filtrabili:

- ☐ **Same** = Come sopra.
- ☐ **Increased** = Valore aumentato. Cerca tutti gli address di cui il valore è maggiore di quello precedente.
- ☐ **Increased By** = Con questa è possibile inoltre specificare di quanto è aumentato il valore dell'address.
- ☐ **Decreased** = Valore diminuito. Cerca tutti gli address di cui il valore è minore di quello precedente.
- ☐ **Decreased By** = Con questa è possibile inoltre specificare di quanto è diminuito il valore dell'address.
- ☐ **Changed** = Valore cambiato. Può essere sia aumentato che diminuito.
- ☐ **Unchanged** = Valore non cambiato. Uguale a prima.
- ☐ **First Scan** = Lo scan successivo verrà comparato con il primo scan della memoria, e non con l'ultimo effettuato.

3.4 – Prima scansione

Adesso apriamo invece *Prato Fiorito* di Windows (che tutti dovreste avere) e con CE apriamo il processo (dovreste già saperlo fare. Il nome del processo è solitamente *MineSweeper.exe*).

Adesso ipotizziamo che noi vogliamo cercare l'address che contiene come valore il tempo trascorso. Per fare ciò dobbiamo prima di tutto avviare il gioco. Clicchiamo su una casella qualsiasi su *Prato fiorito*, e il tempo comincerà a scorrere. Ora su CE in basso a sinistra clicchiamo su

e poi nella nuova finestra clicchiamo sul simbolo di pausa in alto a sinistra per mettere in pausa il gioco. A me il gioco si è fermato con il tempo a 54 secondi, quindi, tenendo il gioco in pausa, inserirò nella TextBox il numero 54 (voi dovete inserire il vostro valore), come Scan Type metterò perché voglio trovare tutti gli address che hanno proprio quel valore e come Value Type sceglierò perché non so in che tipo è rappresentato il tempo. Poi faccio click su First Scan. Al termine mi verranno mostrati una serie di address nella lista a fianco. Ora facciamo scorrere il gioco, riprendendo di nuovo sul bottone che avevamo usato per metterlo in pausa, e rifermiamolo dopo qualche secondo. Il tempo smetterà di nuovo di scorrere, adesso a me si è fermato su 58. Adesso possiamo fare:

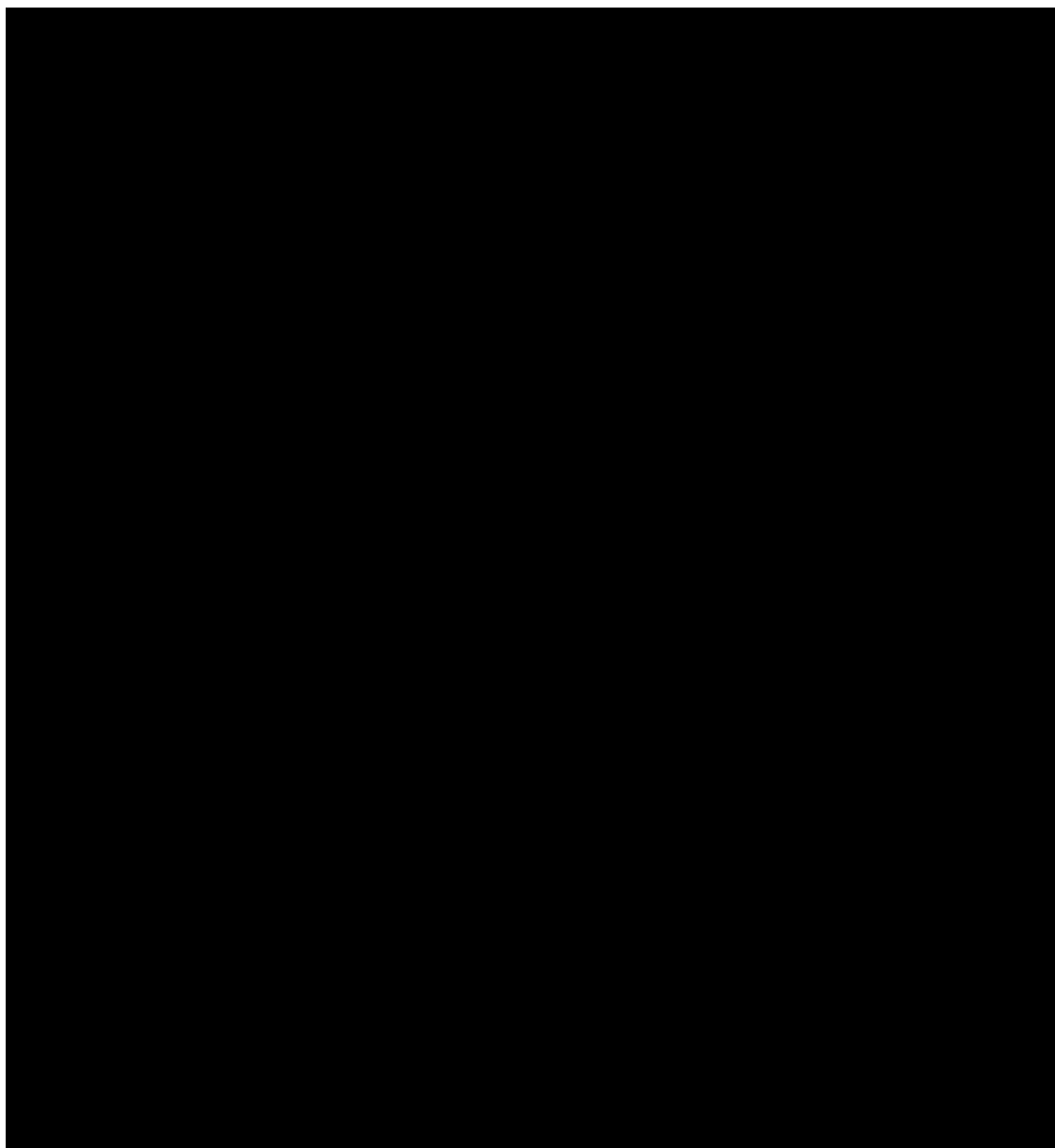
- Un altro perché sappiamo che il valore è adesso 58 (per voi sarà diverso).
- Un perché sappiamo che il valore è aumentato di 4 (per voi sarà diverso).
- Un perché sappiamo che il valore è aumentato. Questa scelta è però sconsigliata perché è meno precisa.

Per cambiare useremo e inseriremo quindi nella TextBox la differenza tra il nuovo valore del tempo e quello precedente (nel mio caso: $58 - 54 = 4$). Poi facciamo click su

. Alla fine rimarrà solo 1 address nella lista di fianco. In particolare a me è rimasto:

004A1470 58

A voi sarà sicuramente diverso. Comunque fate ora doppio click su quell'address e comparirà nello spazio in basso. Risultato:



Come vedete è di tipo Float (anche chiamato Single). Adesso facciamo scorrere il gioco ripremendo sul solito bottone con il simbolo di pausa. Con lo scorrere del tempo anche il valore dell'address aumenterà. Adesso se facciamo doppio click sul valore (oppure clicchiamo con il destro e facciamo *Change record Value*) ci comparirà una finestra dove è possibile modificare il valore. Mettiamo ad esempio 1 e premiamo su OK. Come vedete il timer del gioco si è azzerato e ha ricominciato a scorrere da 1. In questo modo abbiamo riguadagnato tempo. Adesso come facciamo ad evitare che il valore aumenti? C'è un modo molto semplice, basta cliccare sulla casella a sinistra dell'address. In questo modo l'address viene e non potrà più aumentare. Possiamo così terminare la partita *impiegando* solo 1 secondo ;) ecco un buon modo per sorprendere gli amici.

3.5 – Dynamic Movement Address

Il `mov` è lo spostamento degli address a ogni avvio del gioco o quando succede qualcosa di particolare. Un esempio è Metin2 dove ogni volta che il gioco viene riavviato, la posizione dei vari valori cambia. Ad esempio se prima il valore della velocità di movimento si trova all'address 004A1463 poi dopo il riavvio si troverà ad esempio all'address 004E5B14. Sembra quindi che non sia possibile creare hack, perché se si scrive un codice che modifica il valore di 004A1463, dopo il riavvio del gioco la velocità di movimento si sarà spostata sull'address 004E5B14 e quindi l'hack non funzionerebbe più.

Ma c'è un truccetto. Il gioco sa sempre dove andare a pescare la velocità di movimento, anche se l'address è cambiato. Lo fa utilizzando i `pointers`. I `pointers` non sono altro che indirizzi di memoria (address) che "puntano" ad altri indirizzi. Per esempio:

L'address 004B1432 contiene la velocità di movimento.

Il suo `pointer`, che si trova all'indirizzo 004C2626 conterrà invece come valore 004B1432.

Se poi l'address della velocità di movimento cambia posizione e va all'indirizzo 004D1D53, il suo `pointer` (004C2626) cambierà il proprio valore in 004D1D53, quindi punterà sempre e comunque all'address della velocità di movimento.

(questo era solo un esempio, gli address li ho tirati a caso).

È chiaro quindi che per creare quindi un programma che modifichi un indirizzo in memoria e che funzioni sempre, dobbiamo fare così:

1. Leggere il valore del `pointer`
2. Scrivere il valore dell'address che abbiamo appena ricavato dal `pointer` con il nuovo valore che vogliamo mettere

Inoltre si aggiunge un altro problema: spesso non basta leggere il valore del `pointer` e poi andare a modificare l'address, ma a volte per trovare l'address dobbiamo sommare al valore del `pointer` un altro valore, chiamato `offset`. Ad esempio:

L'address 004B1432 contiene la velocità di movimento.

Il suo `pointer`, che si trova all'indirizzo 004C2626 conterrà invece come valore 004B13D7.

L'offset del `pointer` sarà quindi 5B. Infatti $004B13D7 + 5B = 004B1432$ che è il nostro address.

Quindi per creare un programma che modifichi l'indirizzo in memoria dovremo fare:

1. Leggere il valore del `pointer`
2. Sommare l'offset al valore appena letto dal `pointer`
3. Scrivere il valore dell'address, che abbiamo appena ricavato dalla somma, con il nuovo valore che vogliamo mettere

E c'è ancora poi un altro problema! I `pointers` sono multi-livello. Esatto! Esistono anche `pointers` multi-livello! Non è eccitante?!? Con `pointers` multi-livello si intende ad esempio: un `pointer`, che punta ad un `pointer`, che punta ad un altro `pointer`, che punta ad un altro `pointer`, che punta all'address da modificare. Ci sarà da divertirsi XD.

Inoltre ad ogni `pointer` va quasi sempre sommato un `offset`. Quindi per creare un programma che modifichi l'indirizzo in memoria, supponendo che ci sia un `pointer` di terzo livello (`pointer` `pointer` `pointer` address), dovremo fare così:

1. Leggere il valore del primo `pointer`
2. Sommare il primo `offset` al valore appena letto dal primo `pointer`
3. Leggere il valore del nuovo address, che sarà il secondo `pointer`

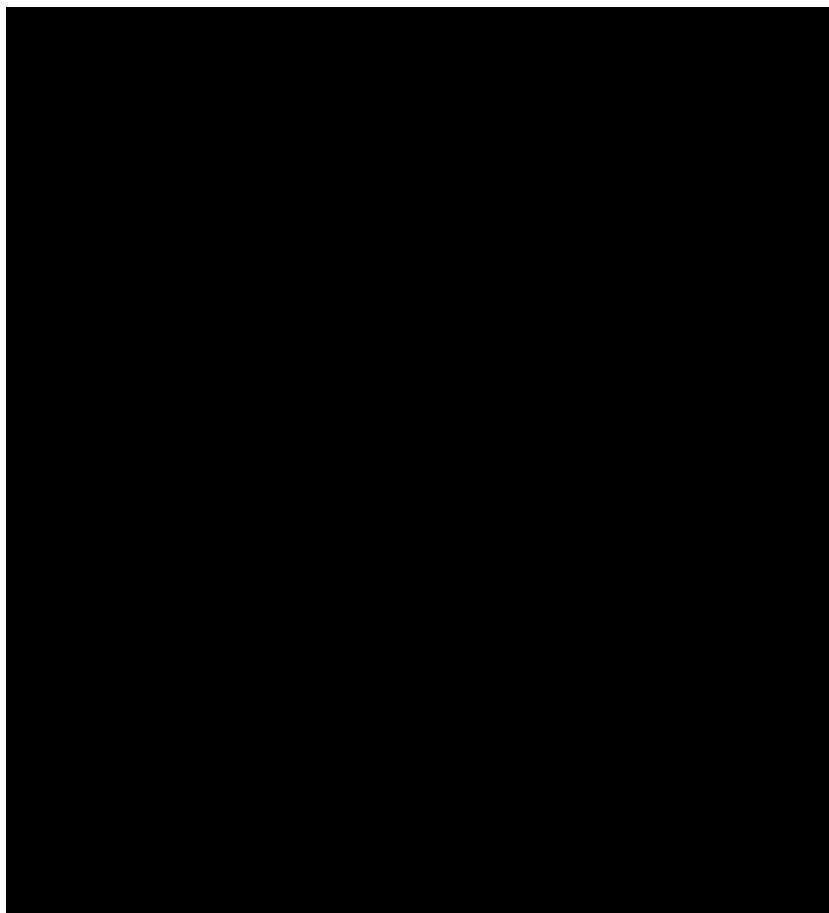
4. Sommare il secondo offset al valore appena letto dal secondo pointer
5. Leggere il valore del nuovo address, che sarà il terzo pointer
6. Sommare il terzo offset al valore appena letto dal terzo pointer
7. Scrivere il nuovo valore che desideriamo nell'address appena trovato

Anche Prato fiorito è protetto da DMA. Infatti se proviamo a chiudere il gioco, a riaprirlo e a riselzionare il processo, vedremo che l'address di prima non conterrà più il valore del tempo di gioco.

3.6 – Pointer scanning

Cheat Engine dispone di un potente [Memory Scanner](#), che ci permette di trovare in breve tempo i pointers di un address.

Sempre con Prato fiorito cercate l'address del tempo (dovreste saperlo fare da soli a questo punto), e fate doppio click su di esso per copiarlo in basso. Adesso, lasciando scorrere il gioco, cliccate con il tasto destro sull'address in basso e poi su [Memory Scanner](#), e ci comparirà questa finestra:

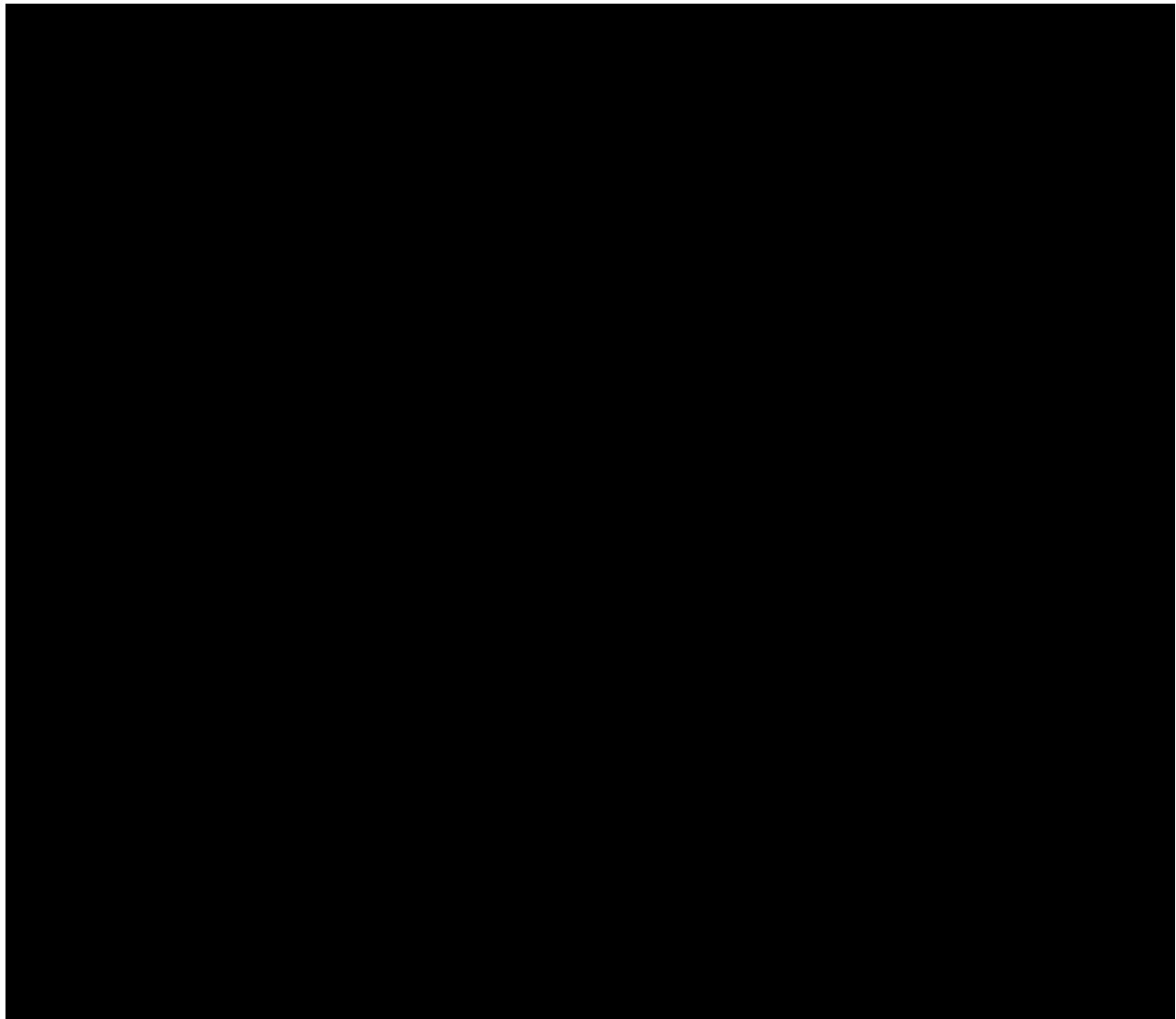


Qui possiamo modificare varie opzioni. Quelle più importanti sono:

Nr of threads scanning = un thread è la suddivisione di un processo (in questo caso il processo di Cheat Engine) che lavora simultaneamente ad un altro thread (o più thread). Quindi più il numero di thread è alto, più la scansione sarà veloce, ma appesantisce il lavoro alla CPU. La ComboBox accanto non so se serve a decidere se CE può usare più o meno del numero di thread specificato oppure una specie di velocità di scansione. Solitamente io metto *Higher*.

Maximum offset value = è chiaro. È il massimo valore che gli offset cercati devono avere. 2048 è perfetto.

Max level = il numero massimo di livello del pointer (il livello dell'offset sarà lo stesso del pointer, ovviamente). Noi impostiamolo su 2, dovrebbe essere sufficiente. Premete su OK, salvate il file dove volete e aspettate. Dopo qualche secondo o minuto comparirà una schermata come questa:



Tutti quelli sono pointers di doppio livello. Noi ne prenderemo uno a caso, ad esempio il primo, e ci scriviamo tutti i dati su un foglio:

Pointer = "minesweeper.exe"+000AAA38

Offset0 = 18

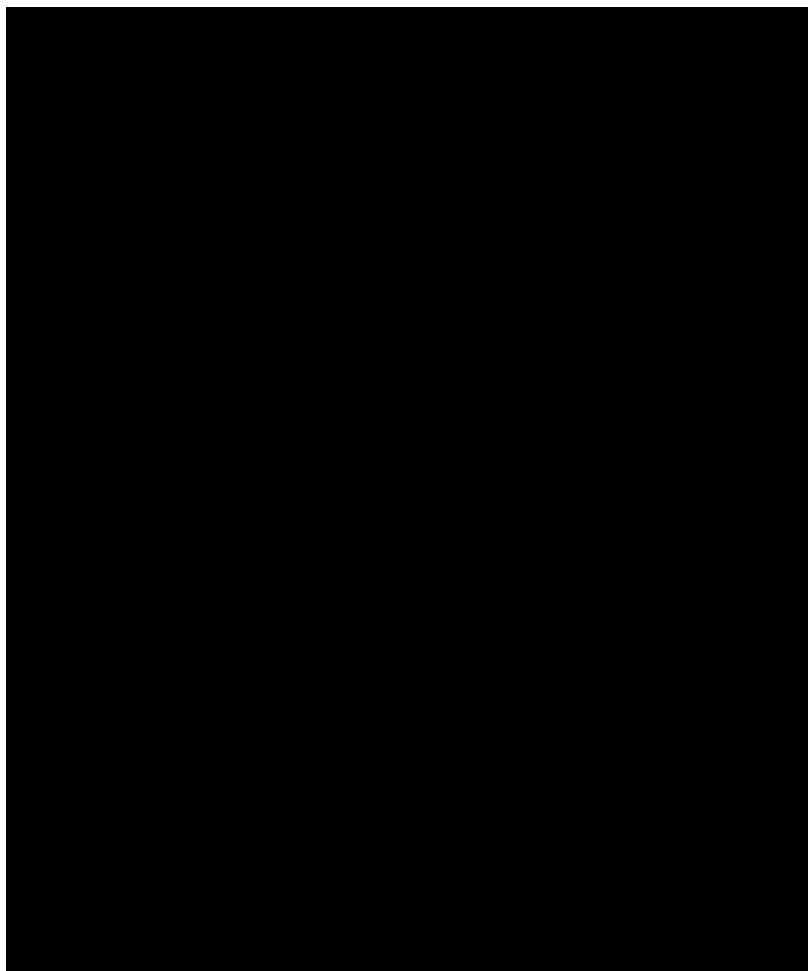
Offset1 = 20

Ora c'è quel minesweeper.exe che può sembrare strano. Quello indica il XXXXXXXXXX del processo. Vi risparmio la spiegazione di cosa si tratta, si trova nella guida all'assembly che vi ho passato qualche paragrafo indietro, e se volete saperlo vi leggete tutta la guida. Vi dico solo che, spesso, il BaseAddress di un processo è XXXXXXXXXX. Quindi il pointer vero e proprio sarà:

$400000 + \text{AAA38} =$

Questo è un modo per trovare i pointer. Il pointer scanner di CE però non sempre funziona (in realtà funziona sempre, ma a volte il BaseAddress non è 400000_{16}), in questo caso no, infatti quello che abbiamo trovato non è il nostro pointer. Infatti se si andasse a creare l'hack con i dati che abbiamo non funzionerebbe. Quindi, a volte, per trovare i pointer dobbiamo appoggiarsi al meto-

do del debugger (anche se il pointer scanner funziona, questo secondo metodo può essere utile molte volte). Chiudiamo il pointer scanner e nella schermata principale di CE, clicchiamo di nuovo con il destro sull'address del tempo ma questa volta facciamo (ovvero trova tutte le istruzioni ASM del programma che vanno a modificare tale address). Alla domanda che compare, che ci avverte che entrerà in funzione il debugger di CE, premiamo Yes. Ci comparirà una schermata simile alla seguente:



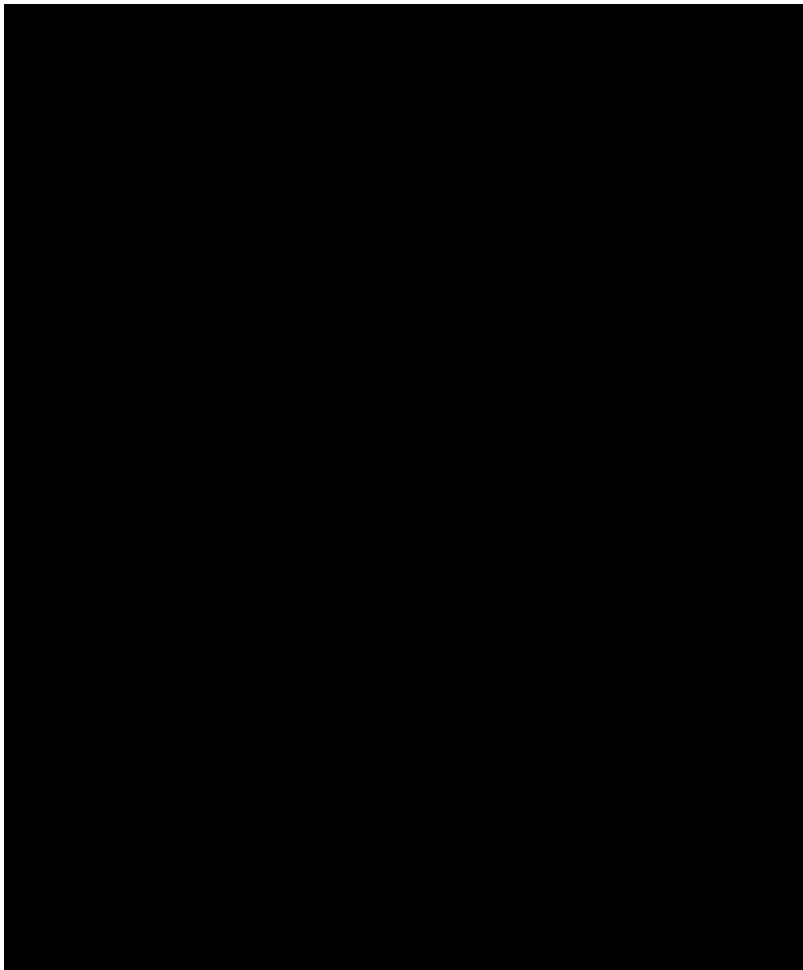
Qui sono utili le conoscenze dell'ASM, comunque selezionatene una qualsiasi (se avete conoscenze di ASM non sbaglierete la scelta) e poi cliccate su *More information*.

Nella schermata che compare troverete la scritta:

The value of the pointer needed to find this address is probably 002FFAC0

Io ho 002FFAC0 ma voi potreste averlo diverso. Ora torniamo alla schermata principale di CE, facciamo *New Scan*, selezioniamo come tipo *4 Bytes* e come scansione *Exact Value* e poi spuntiamo la casella ☐ vicino alla TextBox, perché ora dovremo cercare un valore esadecimale. CE ci ha detto il probabile valore del pointer, quindi noi adesso cercheremo il pointer manualmente. Nella TextBox mettiamo 002FFAC0 (voi mettete il vostro valore, io ho questo) e facciamo *First Scan*.

A me sono spuntati 3 address tutti con lo stesso valore. Seleziono il primo e provo a fare *Find out what accesses this address* su questo nuovo address... Non succede niente. Provo con il secondo e... mi compare la schermata:



Seleziono la seconda e faccio *More information*. E questa volta mi dice che il valore del pointer di secondo livello è probabilmente 002FACE0 (a voi è probabilmente diverso), quindi ripetiamo di nuovo la ricerca con il nuovo valore. Questa volta mi sono usciti ben 9 address. Io scelgo il primo, che è 001FF6F8.

Allora abbiamo:

Pointer 1 = 001FF6F8 che contiene come valore 002FACE0

Pointer 2 = 002FACF8 che contiene come valore 002FFAC0

E l'address da modificare che è 002FFAE0

Ora si può ricavare il primo offset facendo: $002FACF8 - 002FACE0 = 18$; e il secondo offset facendo: $002FFAE0 - 002FFAC0 = 20$.

Abbiamo così trovato i dati che ci interessano. In realtà quello che ho appena trovato non è un pointer statico, bisognerebbe ancora continuare e salire di livelli ma dovrete aver capito come funziona.

Unità 4

Code-Cave

- Cos'è il code-cave
- Il disassembler di Cheat Engine
- Primo code-cave

4.1 – Cos'è il code-cave

Un (o) può essere definito come la modifica, anche a (ovvero durante l'esecuzione del programma/gioco), di una parte del codice del programma/gioco che si trova in memoria. Solitamente per il code-cave si usano le istruzioni assembly JMP e CALL, che permettono di saltare ad un'altra parte del codice del programma, in modo tale da modificare l'esecuzione. Viene però spesso usata anche l'istruzione NOP, che indica "non fare nulla", per annullare l'esecuzione delle istruzioni. Ora vi farò un esempio (non in ASM ma in pseudo-linguaggio perché non so se conoscete l'assembly) di code-cave. Questo è un possibile blocco di codice che può aumentare il timer di un gioco:

```
...istruzioni precedenti
Prendi il valore attuale del timer
Aumenta il valore di 1
Setta il nuovo valore del timer
...istruzioni successive
```


Noi vogliamo fermare il timer

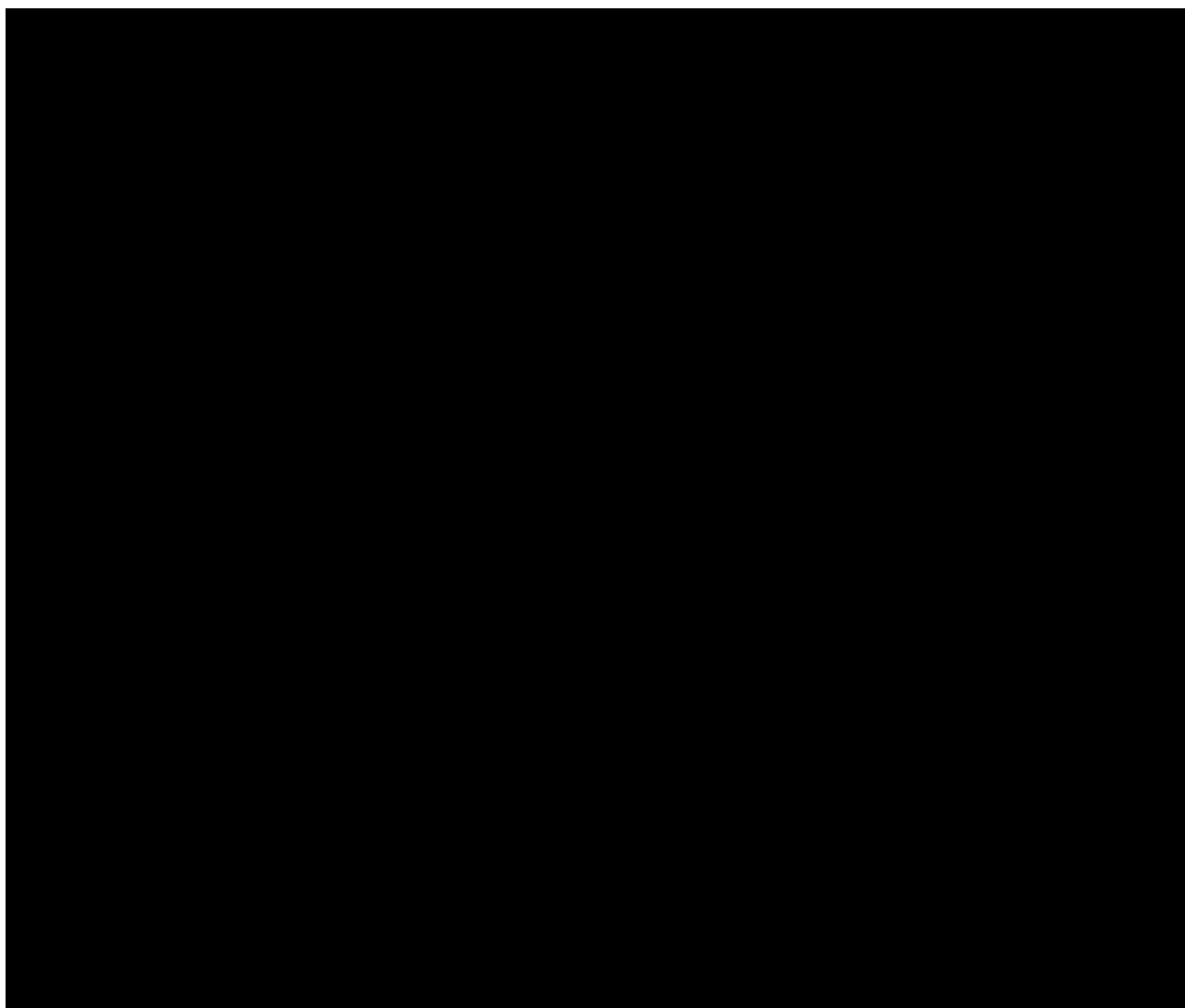
Allora si potrebbe fare un più modi:

- 1- Modificare la riga "Aumenta il valore di 1" con un NOP, così il timer si arresta.
- 2- Inserire un JMP al posto di "Prendi il valore attuale del timer" che salti all'indirizzo che contiene l'istruzione successiva a "Setta il nuovo valore del timer", così da saltare tutto il blocco di codice.
- 3- Effettuare una CALL (prima o dopo il blocco) che porti ad un codice (creato da noi in un'area di memoria libera) che abbassa il timer di 1, così resta fermo.

Il code-cave è un'arma molto potente e pericolosa. Pensate che viene utilizzata da molti virus, soprattutto di tipo *Worms*, per infettare altri file del computer e creare delle copie di se stesso. Infatti se ci pensate bene, se si modifica l'esecuzione di un programma è possibile creare un altro virus che poi si troverà nascosto all'interno del programma infetto. Ma non è questo lo scopo per cui vi insegnerò a fare un code-cave.

4.2 – Il disassembler di Cheat Engine

Un disassembler è un software in grado di tradurre il linguaggio macchina in linguaggio assembly. Dovete prima di tutto aver chiaro che in memoria non si trovano le istruzioni assembly, ma una serie di bit (01001110....) che nel loro insieme corrispondono a delle istruzioni ASM. Ad esempio l'istruzione NOP corrisponde al valore esadecimale 90 che a sua volta corrisponde a 10010000 binario. In memoria si troverà scritto quindi 10010000 e il disassembler ci mostrerà invece NOP. Un disassembler ci traduce quindi i bytes contenuti in memoria nelle istruzioni assembly. Anche CE dispone di un disassembler. Per vederlo aprite un processo qualsiasi e poi fate click sul bottone . Si aprirà una schermata simile alla seguente:



La finestra è divisa in due parti principali: il disassembler (posizionata in alto) e l'editor esadecimale (posizionato in basso, quello con le scritte verdi). Noi lasceremo da parte l'editor esadecimale e lavoreremo solo con il disassembler. Ora analizziamo la parte del disassembler: esso è diviso in varie colonne:

- Qui viene visualizzata l'address che contiene l'istruzione.
 - Qui vengono visualizzati, sotto forma esadecimale, i bytes contenuti in memoria. Vediamo che la prima riga contiene FC. FC è infatti il corrispondente valore esadecimale dell'istruzione cld (Clear Direction Flag) contenuta in quell'address.
 - Visualizza il contenuto dell'address tradotto in linguaggio assembly.
 - : Contiene i commenti.
- : Per una lista di tutti gli opcodes (istruzioni ASM) e i loro corrispondenti valori esadecimali, basta aprire il file in allegato a questa guida. Questo può esservi molto utile.

4.3 – Primo code-cave

Questa volta noi vogliamo far sì che il timer del solitario di Windows scorra al rovescio. È un code-cave semplicissimo. Ecco come faremo:

Utilizzeremo il metodo del JMP. Ovvero cercheremo quell'istruzione che fa aumentare di uno il timer del gioco, la modificheremo inserendoci un JMP ad uno spazio di memoria libero, poi in questo spazio di memoria libero inseriremo un codice che faccia invece diminuire di uno il timer del gioco, poi nell'istruzione successiva a questa inseriremo un altro JMP che salti all'istruzione successiva a quella che aumentava il timer, così da far continuare regolarmente il gioco.

Allora passiamo alla pratica:

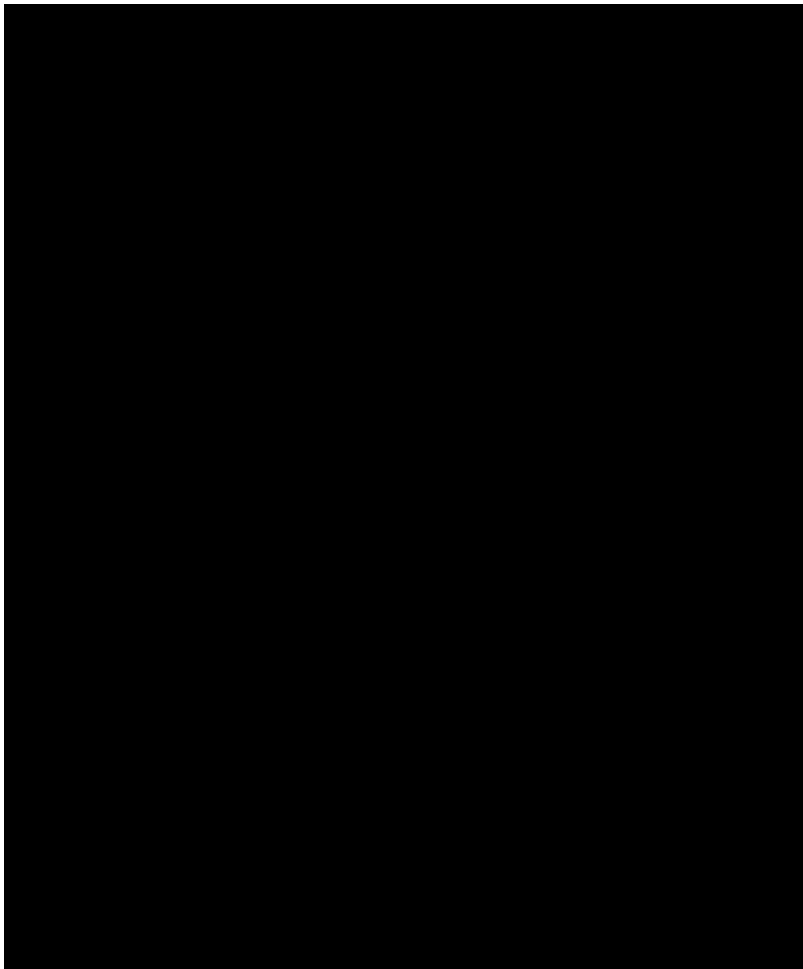
Aprirete il solitario e avviate la partita facendo click sul mazzo. Il tempo comincerà a scorrere. Ora impostate come tipo di valore 4 bytes e cercate (come si faceva con prato fiorito) l'address che contiene il tempo trascorso. A me alla fine rimangono 2 address:

000EEC20

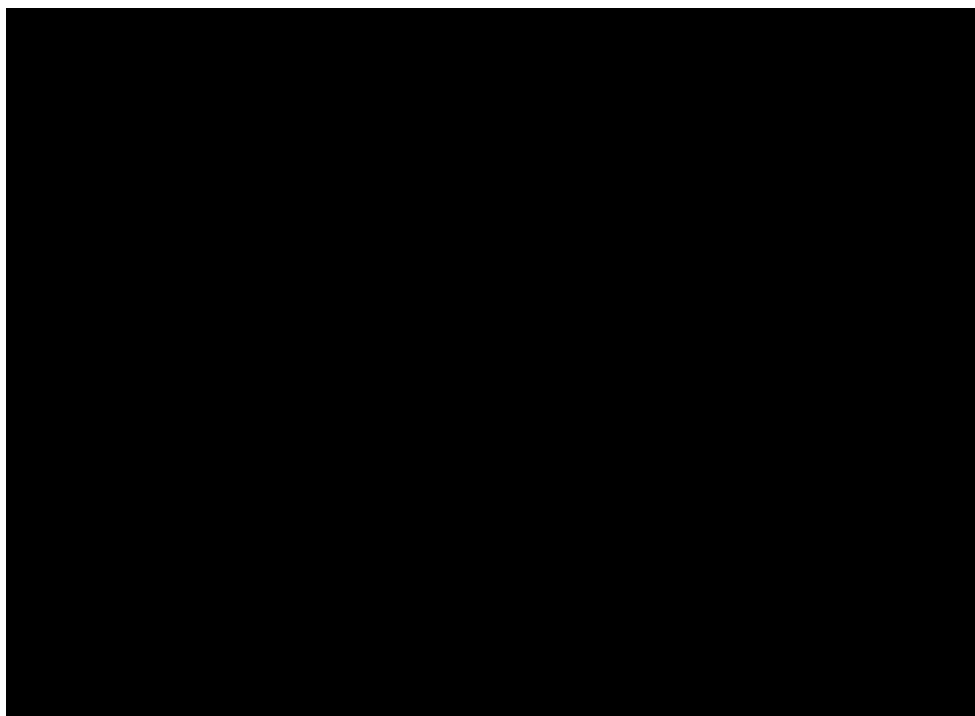
0023A5CC

Scelgo il secondo perché per varie ragioni mi ispira più fiducia :D.

Faccio doppio click su di esso per portarlo in basso. Adesso click destro su di esso e *Find out what accesses this address*. Mi compare questa schermata:



Notate: l'address FF259681 modifica il valore con l'istruzione `inc[rax+0C]`. Se conoscete un po' di assembly, sapreste che l'istruzione `inc` aumenta di uno l'address (sembra proprio l'istruzione che incrementa il timer). In questo caso aumenta di uno l'address `[rax+0C]`. ma cos'è `[rax+0C]`? `rax` è un registro e `0C` è un valore esadecimale. Anche i registri possono essere visti come particolari indirizzi di memoria. Quindi quel `rax` non è altro che il pointer dell'address del tempo di gioco, e quell'`0C` non è altro che l'offset del pointer. Selezioniamo quell'istruzione e facciamo *More information*. Comparirà:



Questa pagina ci mostra il valore di `rax`, ovvero 0023A5C0. Proviamo a sommarci l'offset 0C:

`23A5C0 + 0C =`

Che è proprio l'address che contiene il valore del tempo di gioco, quello che avevamo trovato all'inizio.

Pare chiaro adesso che è proprio come pensavamo. Quella che abbiamo trovato (`inc [rax+0C]`) è proprio l'istruzione che aumenta di uno il valore del timer. Adesso nella finestra prima riselectiamo tale istruzione e clicchiamo su `More information`. Si aprirà il disassembler proprio sull'istruzione `inc [rax+0C]`. Ora facciamo click sul menu *Tools* e poi su *Auto assemble*. Nella finestra che compare facciamo click su *Template* e poi *Code Injection*. Nella finestra che compare fate click su OK e poi compariranno delle scritte simili a queste:

Allora CE ci divide il codice in varie aree. Quelle che ci interessano sono newmem e originalcode. In originalcode c'è l'istruzione inc [rax+0C]. newmem è invece l'area di memoria libera dove metteremo l'istruzione di decrementare di uno il timer. Aggiungo che l'area "solitaire.exe"+39681 mostra come verrà modificata la parte originalcode dopo il code-cave: con un salto a newmem seguito da tre NOP. Quindi cancelliamo l'istruzione inc [rax+0C] da originalcode e sotto newmem inseriamo l'istruzione dec [rax+0C] (dec: decrementa di uno). Poi clicchiamo su *Execute*, facciamo Yes alla finestra che compare (ci chiede se siamo sicuri di eseguire il code-cave), e premiamo su OK al messaggio di conferma che ci compare dopo. Guardiamo adesso sul solitario e... BOOM! Il tempo scorre al rovescio!

i io r n m i go chg l'istruzioone g c [rem l à

Unità 5

Ripartiamo e approfondiamo!

- Perché questa Unità?
- Open process
- Finestra principale
- Finestra del pointer-scanner
- Finestra del debugger
- Finestra del dissassembler e dell'hex editor
- Conclusioni

5.1 – Perché questa Unità?

Questa guida è stata scritta per la sezione VB / VB.NET di InForge, per spiegare come creare hack in vb.net. Poi ho ripreso una parte della guida (quella che avete appena letto) togliendo la parte

5.2 – Open process

Abbiamo imparato ad aprire un processo cliccando sul computerino in alto a CE, selezionando il processo, e poi cliccando su *Open*. CE ci mette però a disposizione altre “modalità” di apertura di un processo. Analizziamole:

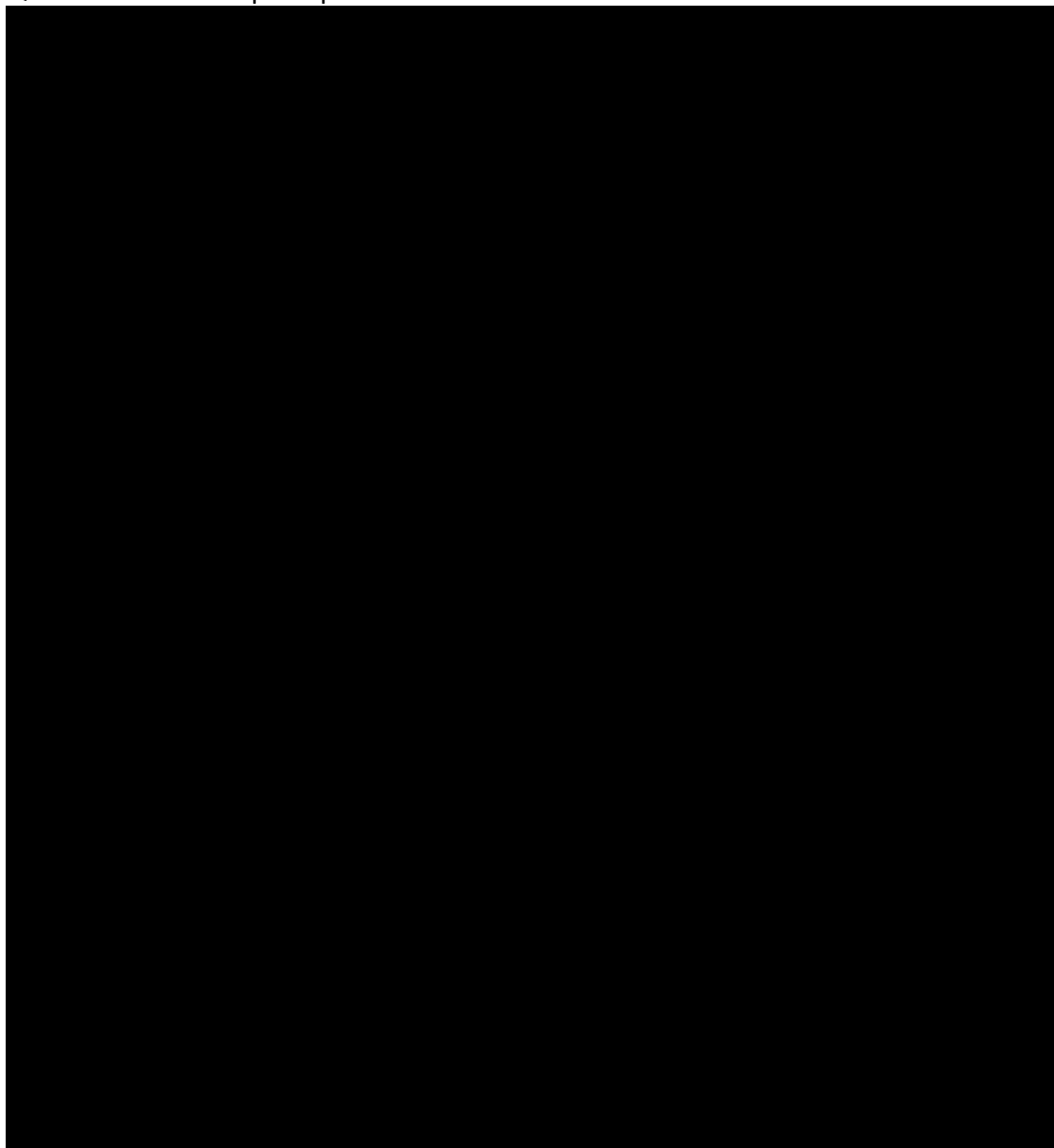
- attacca il debugger al processo, senza aprilo.
- crea un nuovo eseguibile che verrà scritto con CE direttamente con il disassembler. In poche parole, crea un processo vuoto.
- apre un eseguibile, lo carica in memoria e ne apre il processo. La differenza dall'Open normale è che quest'ultimo apre un processo già caricato in memoria.

Ci sono poi altre funzioni particolari:

- impostata per default. Visualizza un lista dei principali processi.
- visualizza la lista completa dei processi.
- visualizza una lista delle finestre, non dei processi.

5.3 – Finestra principale

Questa è la finestra principale di CE:



Vediamo cosa ci offre:

- pulisce la lista degli address e si prepara per una nuova scansione.
- inizia la prima scansione.
- filtra i risultati trovati nella precedente scansione secondo i criteri stabiliti.
- torna alla scansione precedente.
- indica che il valore contenuto nella TextBox è un valore esadecimale.
- TextBox di ricerca.

- impostazioni generali di Cheat Engine.
- tipo di ricerca.
- tipo di valore.
- - *Start*: address di inizio della scansione. Gli address precedenti a quell'indirizzo non verranno presi in considerazione.
 - *Stop*: address di fine della scansione. Gli address successivi a quell'indirizzo non verranno presi in considerazione.
 - *Writable*: cerca anche nella memoria scrivibile (spiegherò più avanti).
 - *Executable*: cerca anche nella memoria eseguibile (spiegherò più avanti).
 - *Fast Scan*: permette di velocizzare la scansione saltando gli address non allineati secondo il numero di bytes specificati (vedere guida all'assembly della Ra.M. Software).
 - *Pause the game while scanning*: mette in pausa il gioco durante la scansione.
- cerca i codici che generano valori casuali. Se li trovi li disabilita. Questa è da usare solo in caso di vera necessità perché CE potrebbe disabilitare codici che non devono essere toccati, provocando crash o problemi durante la ricerca degli address.
- aumenta o riduce la velocità di esecuzione del processo, costringendo la CPU a dedicare più tempo a quel processo.
- permette di aggiungere un address manualmente. Ci sono anche funzioni per inserire un pointer (lo abbiamo già visto).
- visualizza l'area di memoria del processo nel disassembler.
- opzioni avanzate dove ce ne una che permette di mettere in pausa il gioco.
- permette di scrivere commenti.

Cerchiamo un qualsiasi valore con CE. Quando ci compariranno i vari address, nello specchietto a sinistra, selezionatene uno e cliccate con il tasto destro. Compariranno le seguenti opzioni:

- visualizza nel editor esadecimale l'area di memoria dell'address selezionato.
- visualizza nel disassembler l'area di memoria dell'address selezionato.
- rimuove dalla lista l'address (o gli address) selezionato/i.

Quando invece facciamo doppio click su un address, esso comparirà in basso. Clicchiamoci sopra con il tasto destro e vediamo cosa possiamo fare:

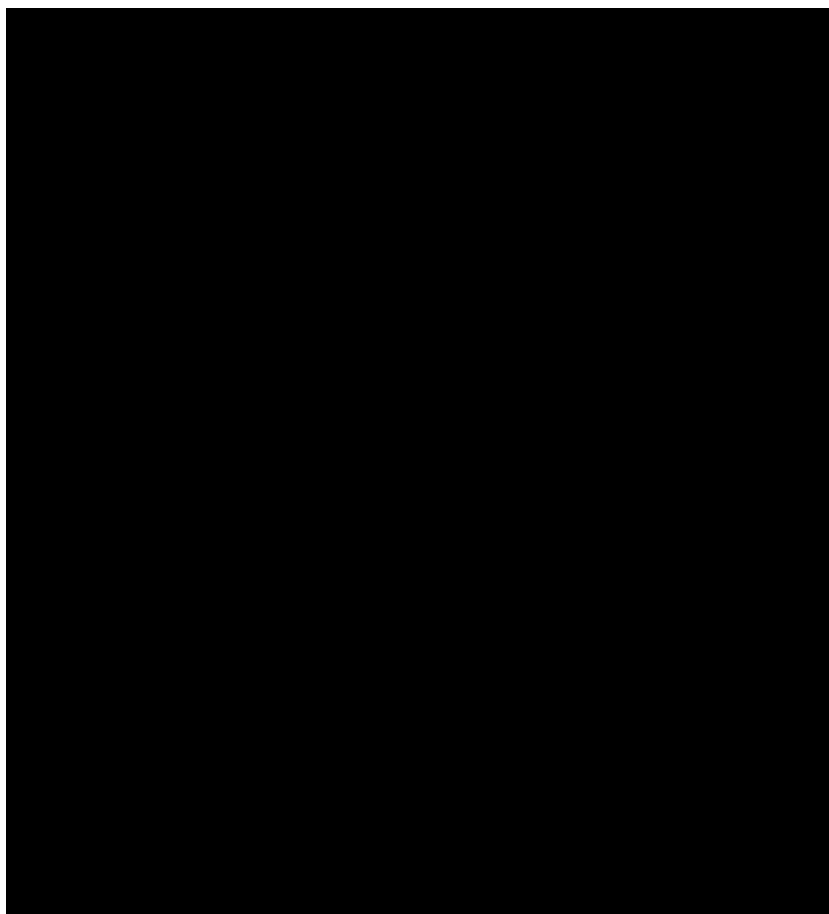
- elimina l'address dalla lista.
- - *Description*: modifica la descrizione dell'address.
 - *Address*: modifica l'address.
 - *Type*: modifica il tipo di valore contenuto dall'address.
 - *Value*: modifica il valore contenuto dall'address.

- *Smart edit address(es)*: modifica le descrizioni di tutti gli address contenenti una determinata descrizione.

- visualizza nel editor esadecimale l'area di memoria dell'address selezionato.
- visualizza il valore dell'address come numero esadecimale/decimale.
- cambia colore all'address.
- aprirà un nuova finestra che permette di impostare delle hotkeys per l'address selezionato. Le hotkeys sono delle combinazioni di tasti della tastiera che quando vengono premuti, eseguono una specifica azione. Vediamo che nella finestra possiamo permettere varie azioni da eseguire sull'address alla pressione di una combinazioni di tasti.
- congela/scongela gli address selezionati.
- apre il pointer scanner (che analizzeremo dopo in dettaglio).
- cerca quali parti di codice accedono (modificano o leggono) questo address (analizzeremo dopo in dettaglio).
- cerca quali parti di codice modificano questo address (analizzeremo dopo in dettaglio).

Adesso analizziamo queste utile tre scelte.

5.4 – Finestra del pointer-scanner



Questa è la finestra di opzioni di scansione. Vediamo che in alto troviamo *Address to find* e *Value to find*. Serve per specificare se deve essere trovato il pointer di un address, o direttamente del valore. Se ad esempio mettiamo address to find, CE cercherà quell'address, poi tutti i pointer che puntano a quell'address. Se invece mettiamo Value to find, CE cercherà quel valore, poi tutti gli address che contengono quel valore e i vari pointer a questi address (molto più complesso).

Poi troviamo:

- gli address devono essere allineati a 32 bit (guardare la guida all'assembly della Ra.M. Software).
- cerca solo i percorsi che hanno un address statico. Ovvero un indirizzo che non si sposta (quei pointer che possiamo usare per le hack, appunto perché non si spostano).
- non include i pointer che hanno dei valori che possono essere solo letti e non modificati.
- solitamente è disabilitata ma io consiglio di abilitarla. Ferma la lista di pointers multi-livello quando trova un pointers statico (che quindi può essere utilizzato nella creazione di un'hack).

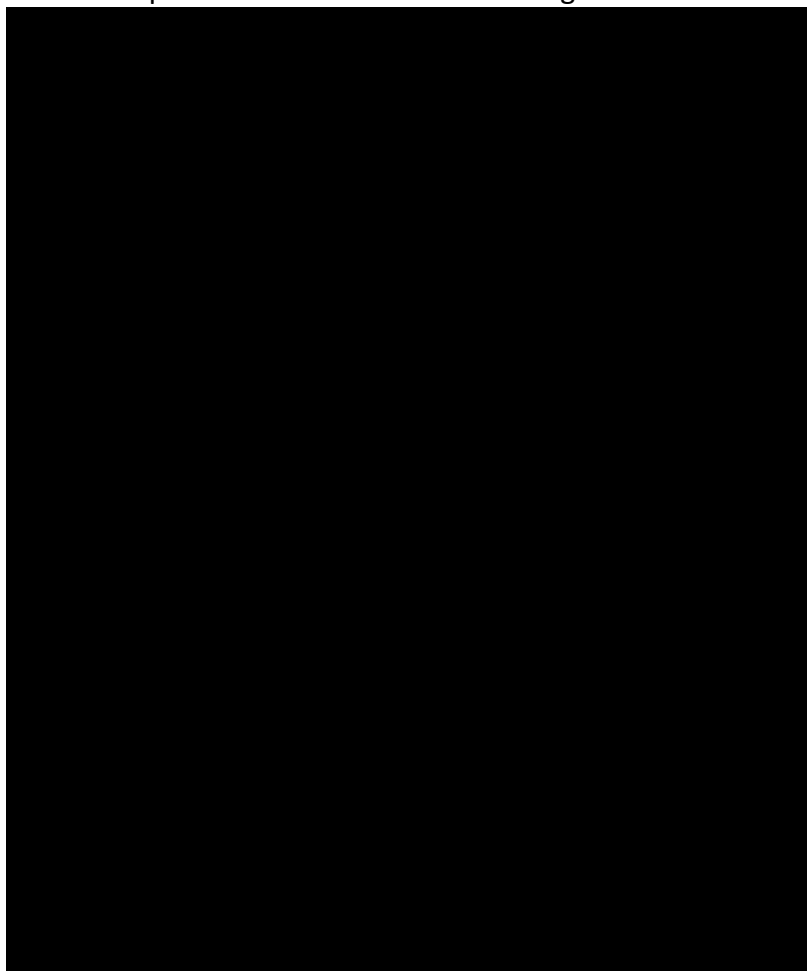
- tutti i pointers devono trovarsi all'intero dell'area di memoria specificata.
- i pointer devono avere gli offset specificati.

Dopo aver terminato la scansione, ci comparirà la lista dei pointers. In alto possiamo settare come visualizzare il valore a cui puntano i pointers (4 Byte, Float o Double).

5.5 – Finestra del debugger

Scegliendo un address dalla lista in basso della finestra di CE, tra le varie opzioni troviamo anche *Find out what accesses this address* e *Find out what writes to this address*.

Queste due funzioni hanno bisogno del debugger per funzionare. Sia con la prima che con la seconda comparirà una finestra simile alla seguente:



In basso, dove vedete scritto “Close”, troverete inizialmente “Stop”. Questo serve a fermare l’analisi del debugger, ovvero ferma il debugger affinché smetta di cercare tutte quelle istruzioni che accedono o modificano l’address scelto.

Nella lista delle istruzioni troviamo due colonne. La prima, è il numero delle volte che quell’istruzione ha modificato l’address (spesso alcuni pezzi di codice, e quindi alcune istruzioni, si ripetono ciclicamente. Questi pezzi di codice vengono definiti `loops`), mentre la seconda, è l’istruzione che modifica l’address. Nell’immagine di esempio la prima istruzione è `inc [rax+0C]`. Per chi conosce l’assembly la capisce subito, comunque significa che incrementa di uno il valore contenuto nell’address puntato dal registro RAX, aumentato di 0C. Pare chiaro quindi che `[rax+0C]` indica l’address stesso che abbiamo scelto. Infatti `inc` modifica quell’address, quindi questa istruzione effettua un accesso in scrittura all’address scelto. Quindi questa istruzione ci comparirà sia che facciamo *Find out what accesses this address* (infatti cerca tutte le istruzioni che accedono, in

lettura o scrittura, a l'address scelto), sia che facciamo *Find out what writes to this address* (infatti cerca tutte le istruzioni che scrivono l'address scelto).

Prediamo invece ad esempio la seconda istruzione: `mov ecx,[rax+0C]`. Questa copia il valore contenuto nell'address puntato da `[rax+0C]` (che già sappiamo che è l'address scelto da noi) nel registro ECX. Di conseguenza effettua solo un accesso in lettura all'address puntato da `[rax+0C]` (il nostro address). Quindi questa istruzione comparirà solo se facciamo *Find out what accesses this address* (infatti si tratta solo di un accesso in lettura, non in scrittura).

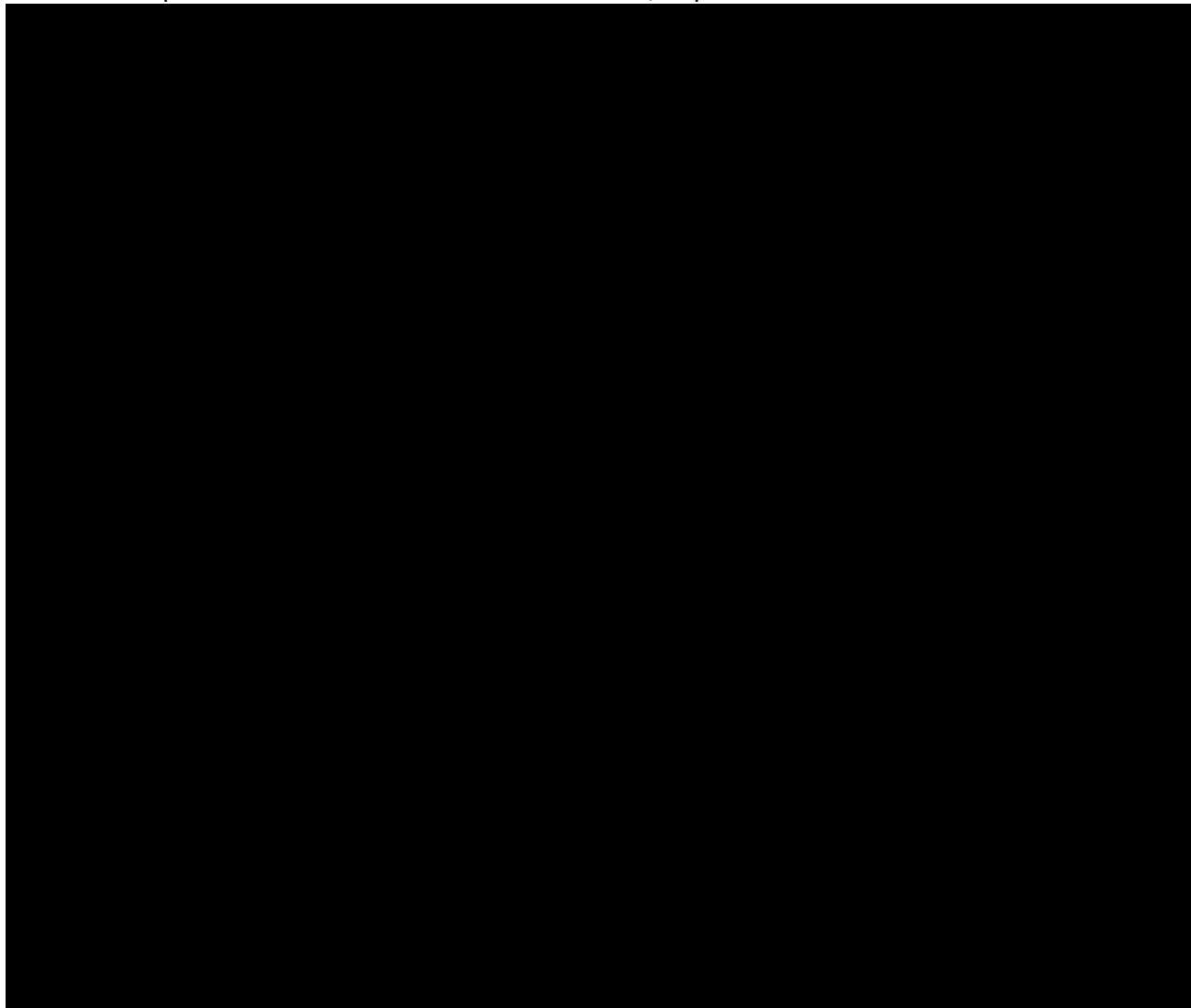
Quando selezioniamo un'istruzione ci compaiono varie scelte:

- sostituisce l'istruzione con un'altra (è necessario conoscere l'assembly) e la copia nella finestra *Advanced Options*.
- visualizza nel disassembler l'istruzione selezionata.
- copia l'istruzione nella finestra *Advanced Options*.
- visualizza informazioni aggiuntive.

Con *Add to the codelist* e *Replace*, il codice viene copiato nella finestra *Advanced Options*. In questa finestra vediamo che possiamo eseguire altre operazioni (intuibili dai nomi) su quelle istruzioni. Nella finestra *More information* troviamo invece informazioni che riguardano: le 5 istruzioni che stanno attorno a quella selezionata, l'effetto dell'istruzione selezionata, il valore del pointer che, probabilmente, punta a questo address, informazioni sui registri generali della CPU, informazioni sulla FPU (F), informazioni sullo stack (S).

5.6 – Finestra del disassembler e dell'hex editor

Cliccando ad esempio su *Show disassembler* si apre la finestra del disassembler e dell'editor esadecimale. La parte alta della finestra è il disassembler, e quella bassa è l'editor esadecimale:



Prima vediamo l'editor esadecimale: troviamo i vari address. I valori o le istruzioni contenute in questi address vengono visualizzati in esadecimale. A destra troviamo anche la codifica ASCII dei valori esadecimali (poco utile). In alto invece troviamo la riga:

Protect: Read/Write Base=012CD000 Size=DF1000 Module=Skype.exe

Protect, indica il tipo di accesso disponibile in quest'area. Read/Write indica che quest'area può essere letta/scritta. Questa è una caratteristica dei sistemi operativi che utilizzano la modalità protetta invece della modalità reale 8086 (leggere la guida all'assembly della Ra.M. Software).

Base, indica il BaseAddress dell'area di memoria (segmento) che stiamo visualizzando (utile quindi per trovare il BA di un pointer).

Size, indica la grandezza dell'area di memoria (segmento) che stiamo visualizzando.

Module, indica il nome del modulo al quale appartiene quell'area di memoria.

Possiamo modificare facilmente i valori esadecimali contenuti. Se invece clicchiamo con il destro, si presentano altre scelte:

- modifica il valore esadecimale selezionato.
- sposta la visualizzazione dell'editor su un altro indirizzo.
- cerca in memoria un testo (ASCII o Unicode) è un array di bytes (espressi in esadecimale).
- come "copia".
- come "incolla".
- cambia la visualizzazione aggruppando o dividendo i bytes.
- aggiunge dei separatori (linea gialla) ogni 2, 4 o 8 bytes.
- tenta di rendere la pagina scrivibile quando non lo è.
- aggiunge un punto di interruzione condizionale sul byte selezionato. Aggiungere un punto di interruzione condizionale significa stabilire che, se una certa condizione si verifica, il processo viene messo in pausa. Possiamo quindi dire a CE di mettere in pausa il processo se accede al byte solo in scrittura o in generale, ecc.

Vediamo adesso il disassembler: è diviso in 4 colonne: la prima, è l'address di memoria da cui comincia l'istruzione. La seconda elenca i bytes che compongono l'istruzione. La terza è l'opcode, ovvero l'istruzione scritta in linguaggio mnemonico (assembly). La quarta contiene i commenti.

Clicca con il destro ci si presentano varie scelte:

- sposta la visualizzazione del disassembler su un altro indirizzo.
- sposta la visualizzazione all'indirizzo precedente.
- sostituisce l'istruzione con un'istruzione NOP (90 esadecimale) che non fa niente.
- assembla il processo con le modifiche apportate.
- come "copia", ma questa volta è possibile scegliere anche cosa copiare.
- cambia i valori dei registri (o dei flags) con nuovi valori scelti dall'utente.
- inserisce un breakpoint incondizionato sull'istruzione selezionata. Quando un breakpoint è incondizionato, significa che ogni volta che quel codice sta per essere eseguito, il processo viene fermato. Inserendo un breakpoint è possibile anche vedere numerose informazioni sui registri e i flags quando l'esecuzione si trova su quella istruzione.
- trova tutte gli address ai quali accede l'istruzione selezionata (il processo inverso di *Find out what accesses this address*).
- un programma è suddiviso in funzioni (sottoprogrammi). Queste opzioni evidenzia tutta la funzione alla quale appartiene l'istruzione selezionata.
- imposta/modifica il commento.

Adesso vediamo i menu in alto (non verranno analizzate tutte le funzioni):

- File
 - apre una nuova finestra del disassembler.
 - salva in un file il codice disassemblato.
- Search
 - cerca un valore testo (ASCII o Unicode) o un array di bytes (in esadecimale).
 - cerca un codice assembly.
- Stack
 - visualizza informazioni che riguardano lo stack del programma (vedere guida all'assembly della Ra.M. Software).
 - visualizza la lista di tutti i breakpoints impostati.
 -

5.7 – Conclusioni

Con questo capitolo finisce la mia guida. Spero vi sia stata utile!! L'utente **Error218** rilascerà, probabilmente, a breve, su InForge, un tutorial eseguibile dove è possibile verificare anche le proprie capacità con CE. Appena verrà rilasciato, consiglio vivamente di scaricarlo ;)

Saluti,

[SpeedJack](#)



[GUIDA] Processi e memoria in Visual Basic .NET by SpeedJack is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/)

Based on a work at www.inforge.net

