

SoftICE

1 INTRODUZIONE

Salve gente, in questo tutorial parlerò di NuMega **SoftICE**, il miglior debugger per piattaforme Win32 attualmente in commercio.

SoftICE è nato come un normale debugger, quindi il suo ruolo principale è quello di agevolare la ricerca degli errori su applicativi di cui si possiedono i sorgenti, ma a "causa" delle sue caratteristiche avanzate è anche lo strumento principale per il reverse engineering.

Il tutorial è sostanzialmente diviso in due parti [SoftICE Basic](#) e [SoftICE Avanzato](#), a coloro che siano alle prime armi consiglio di provare ad utilizzare le conoscenze acquisite tramite la lettura della prima sezione prima di intraprendere la lettura della seconda parte.

I comandi descritti funzionano con le versioni di **SoftICE 3.x** o superiore, è inutile dire che prima di usare **SoftICE** bisogna conoscere l'assembler. Un ultimo consiglio acquistate **SoftICE** così avrete i manuali con tutti i comandi disponibili, ed i mitici programmatori della NuMega potranno guadagnare qualche spicciolo ;)

1.1 Configurare **SoftICE**

Una volta installato **Softice**, dal menu edit selezionare *Softice initialization settings*, troverete:

- o *Initalization String*: sono una serie di comandi che sono eseguiti all'avvio di **Softice** (il ";" indica la pressione del tasto Invio).
- o *Total RAM*: bisogna inserire qui la quantità di RAM nel sistema.
- o *History, Trace, Video buffer size*: indica la grandezza del buffer di History (usato nel command window), trace (usato nel back-trace), video (la RAM video).
Per quanto riguarda l'initialization setting consiglio: `wl;wr;wc 10;wd 3;faults off;x;` (questi comandi saranno spiegati successivamente) mentre per le altre opzioni lasciatele invariate, ad eccezione del Buffer RAM e Video se quelli specificati non coincidono con il sistema.

Ora possiamo cliccare su Ok ;)

Adesso se possedete **Softice** per win9x andate nella dir in cui lo avete installato ed editate il file WINICE.DAT, troverete delle linee di questo tipo:

```
;EXP=c:\windows\system\kernel32.dll
```

eliminate il ";" da tutte le linee che iniziano con EXP e vi sarete risparmiati un po' di lavoro.

Ora abbiamo configurato Softice possiamo riavviare il computer ed iniziare ad imparare!

2 **SoftICE BASIC**

Ok, ora premete CTRL-D ;) hehe siamo in **SoftICE**. Iniziamo dalla schermata che vedete (se avete fatto quello che vi ho detto :P). In alto abbiamo la schermata dei registri (registry window), dove ci sono tutti i registri (sapete che sono vero?), i flags, e sotto i flag - solo per alcune istruzioni - c'è un indirizzo nel formato selettore/offset e la doubleword cui punta (questo è mooolto utile!). Qualche parola in più voglio sprecarla solo per i flags, ci sono dei simboli essi indicano:

```

O D I S Z A P C
| | | | | | | +----- Carry Flag
| | | | | | | +----- Parity Flag
| | | | | | | +----- Auxiliary Carry Flag
| | | | | | | +----- Zero Flag
| | | | | | | +----- Sign Flag
| | | | | | | +----- Interrupt Flag
| | | | | | | +----- Direction Flag
+----- Overflow Flag

```

Se un flag è settato (impostato ad 1) la lettera corrispondente sarà in aiuscolo, altrimenti in minuscolo, ad esempio se abbiamo questa configurazione: "oDisZapc" allora solo il Direction Flag e il Flag Zero sono settati, quindi in un'istruzione movsb, EDI decrementerà e in un'istruzione JZ la cpu eseguirà il salto.

Più in basso vi è una finestra dati (DATA WINDOW), usata per dare uno sguardo nella RAM del PC.

Poi abbiamo la finestra di codice (CODE WINDOW) dove vi è il codice dissassemblato.

Infine possiamo vedere la finestra dei comandi (COMMAND WINDOW) dove ci sono tutte le informazioni di debugging che ci mette a disposizione **SoftICE**, dove possiamo scrivere i nostri comandi, vedere l'help etc.

I tasti da usare per scrollare queste finestre sono:

ALT-UP/DOWN/PGUP/PGDN:Scrolla il DATA Window

CTRL-UP/DOWN/PGUP/PGDN:Scrolla il CODE Window

SHIFT-UP/DOWN/PGUP/PGDN Scrolla il COMMAND Window

NOTA: per uscire da **SoftICE** basta digitare X e quindi invio oppure premere F5.

Bene, dopo un rapido sguardo a come appare **SoftICE** ora iniziamo ad elencare i comandi più significativi.

2.1 Comandi per il controllo del flusso

In questa sezione presenterò i comandi fondamentali di **SoftICE** quelli relativi al tracing del programma. Per trace del programma s'intende l'esecuzione passo passo delle istruzioni dello stesso. Il tracing viene in Italiano tradotto tracciatura, ma a me non piace molto :D. Essi sono fondamentali perché sono i comandi essenziali di tutti i debugger. La sintassi utilizzata per la specifica dei parametri è quella classica:

- Un parametro tra le parentesi quadre "[]" indica che è opzionale cioè può essere omesso.
- I parametri separati dal simbolo "|" non possono essere specificati contemporaneamente, quindi solo uno dei due può essere utilizzato.
- Gli altri parametri sono essenziali e devono essere specificati.

X

Esce dal debugger e torna al programma corrente. Di solito è associato al tasto "F5".

G [=StartAddress] [BreakAddress]

Continua l'esecuzione del programma corrente da "StartAddress" se specificato altrimenti da CS:EIP (o CS:IP), e successivamente la blocca (torna cioè a **SoftICE**) quando è eseguita un'istruzione all'indirizzo "BreakAddress" se quest'ultimo è specificato. "StartAddress" e "BreakAddress" sono due indirizzi, "BreakAddress" deve puntare al primo byte di un opcode per permettere a **SoftICE** di bloccarsi.

T [=StartAddress] [Count]

Esegue l'istruzione puntata da "StartAddress" se specificato, altrimenti esegue l'istruzione puntata da CS:EIP (o CS:IP). "Count" specifica il numero di istruzioni da tracciare prima di restituire il controllo all'utente. In pratica T serve per tracciare il programma istruzione per istruzione, se è trovata un'istruzione Call o Int, l'esecuzione continua dalla prima istruzione della routine o ISR (Interrupt Service Routine) chiamata. Normalmente il tasto "F8" è impostato in modo da eseguire l'istruzione "T" senza parametri.

P [RET]

Esegue l'istruzione corrente (quella puntata da CS:EIP o CS:IP). Se l'istruzione corrente è una Call, Int, Loop, Rep l'intera routine o iterazione è eseguita e solo allora il controllo torna a Sice. Il parametro "RET" se specificato indica a **SoftICE** di eseguire tutte le istruzioni seguenti fino a che non è eseguita una ret, solo allora il controllo è restituito. In generale al comando "P" è associato il tasto "F10" e questo insieme ad "F8" è usato per tracciare il programma.

HERE

Esegue le istruzioni a partire da quella corrente fino a quella in cui si trova il cursore. Il cursore deve trovarsi nel CODE WINDOW per funzionare.

EXIT

Forza l'uscita dal corrente processo o programma. Nota, a ring 0 non funziona.

HBOOT

Resetta il computer!

.

Riposiziona il codice all'interno della finestra di codice nella posizione dell'istruzione corrente.

Ora qualche esempio chiarificatore. Supponiamo di avere il seguente programma:

```
inizio: mov eax,1
        mov edi,[ebp-4]
        mov ecx, 20
i1: rep scasb
i2: call DoAction
i3: mov edx,1
    add edx,ecx
    xor edx,eax
.....
DoAction: xchg eax,edx
        mov eax,edx
        ret
```

Supponiamo ora di trovarci ad "inizio" e proviamo a tracciare il programma passo passo, con i tasti F8/F10. Se usiamo l'istruzione T (F8), vedremo che l'esecuzione si fermerà + volte nell'istruzione in i1. Usando in quest'istruzione invece il comando P (F10) essa sarà eseguita una sola volta e quando il controllo sarà restituito l'istruzione corrente sarà quella in i2. Se i2 è eseguita utilizzando il tasto F8 l'istruzione successiva sarà quella in DoAction, se è invece eseguita con F10 l'istruzione seguente sarà quella in i3. Facile no!! Niente di complesso fin qui. Il comando G può esserci utile per fermarci ad una locazione specifica ad esempio se ci troviamo ad inizio e vogliamo fermarci all'istruzione in i3 basta digitare il comando G i3, e se ad esempio vogliamo anche saltare le istruzioni prima di il possiamo usare il comando G=i1 i3. Per utilizzare il comando HERE basta posizionarsi con il mouse sull'istruzione in cui vogliamo fermarci e quindi invocarlo, ad esempio se

volgiamo di nuovo fermarci sull'istruzione in i3 basta posizionarsi su quell'istruzione ed eseguire il comando.

2.2 Comandi di gestione dei dati

In questa sezione parlerò dei comandi principali per la gestione dei dati in **SoftICE**, essi sono utilizzati per tener traccia delle modifiche che il programma apporta sui dati.

D [indirizzo [l length]]

Mostra il contenuto della locazione "indirizzo" nella finestra dati, *length* invece ne indica la dimensione. *Indirizzo* può essere scritto nella forma selettore/offset oppure solo offset ed in quest'ultimo caso è utilizzato come segmento di default DS. Per vedere dati di un tipo determinato possiamo usare invece del solo comando "D" un suffisso che ne indica il tipo, più chiaramente:

DB per visualizzare un Byte
DW per visualizzare una Word
DD per visualizzare una DoubleWord
DS per visualizzare una short Real
DL per visualizzare una Long Real
DT per visualizzare una Real da 10 Byte

Per questi comandi si usa la stessa sintassi del comando "D".

E [indirizzo] [data]

Modifica il valore di *indirizzo* con il valore *data*, oppure (se *data* non è specificato) permette di andare in modalità di editing nella finestra dati. Come per il comando "D" c'è la possibilità di usare i suffissi (B,W,D,S,L,T) per editare un dato di un determinato tipo.

DATA [x]

Permette di visualizzare la finestra dati numero "x", dove x è un numero tra 0 e 3. **SoftICE** permette di utilizzare 4 finestre dati che si riferiscono ad altrettanti indirizzi, ma solo uno alla volta può essere visualizzato.

FORMAT

Permette di cambiare il formato della finestra dati corrente. Ad esempio se si stanno visualizzando dati in formato byte e si esegue il comando FORMAT la finestra dati visualizzerà i dati in word, quindi se lo si esegue di nuovo verranno visualizzate delle doubleword e così via.

R[Reg | Reg [=] value]

Permette di modificare il contenuto di un registro. Il Registro *reg* è modificato con il valore *value*.

DEX [WindowNumber [expression]]

Assegna un'espressione alla una finestra dati numero *WindowNumber*. *expression* indica l'espressione che sarà valutata e il cui risultato sarà visualizzato nella finestra dati scelta. La differenza dal comando D è che la valutazione dell'espressione è fatta ogni volta che **SoftICE** esegue un comando di modifica del flusso del codice, mentre l'espressione presente in D è valutata solo prima dell'esecuzione del comando.

S Address L Length str

Cerca la stringa *str* dall'indirizzo *Address* all'indirizzo *Address* + *Length* e ne restituisce gli indirizzi trovati. In alcune versioni di **SoftICE** S restituisce il primo indirizzo dove si trova la stringa e bisogna digitare di nuovo S senza parametri per ricercare la prossima, in altre restituisce una lista di indirizzi in cui è stata trovata la stringa. La stringa *str* è una stringa separata da "" e/o una lista di numeri separati dalla virgola, ad esempio:

```
'GEnius',0,'Ciao',0,3
```

è una stringa valida.

F Address L Length data

Riempie un'area di memoria compresa tra *Address* e *Address + Length* con *data*. *Data* è una stringa compresa tra "" e/o una lista di numeri separati dalla virgola, ad esempio:

```
'GEnius',0,'Ciao',0,3
```

è una stringa valida.

M StartAddress L length EndAddress

Copia il contenuto di un'area di memoria compresa tra *StartAddress* e *StartAddress + length*, nell'area compresa tra *EndAddress* e *EndAddress + length*.

C address1 L length address2

Confronta i byte presenti nelle due area di memoria di lunghezza *length* di indirizzi *address1* e *address2*. Se delle differenze sono trovate, saranno mostrati i byte incriminati ed i relativi indirizzi.

? espressione

Valuta un'espressione e ne visualizza il risultato. La sintassi delle espressioni è simile a quella del C (se non la conoscete cercate qualche tut sul C!). E' molto utile nella visualizzazione dei registri, perché li visualizza in formato esadecimale e decimale, ad esempio provate "? eax".

CONSIDERAZIONI:

Ok ragazzi, dopo che avevate letto tutti questi comandi, avrete un po' le idee confuse, quindi facciamo un po' di esempi per rendere tutto un po' più umano ;). La prima cosa importante da capire è che avete a disposizione 4 DATA WINDOW, numerate da 0 a 3, di cui una sola è visualizzabile in un determinato istante. Quindi se avete bisogno di monitorare 2 o più locazioni di memoria dovete usare il comando DATA per switchare tra le finestre dati. Altra cosa importante è la possibilità di modificare i registri, un'arma indispensabile in molti casi, infine dovrei parlare delle capacità di ricerca, copia, e controllo dei dati di Sice ma penso che il loro uso sia banale ed intuibile, quindi non ne parlo. Ora qualche esempio per schiarirci un po' le idee.

Supponiamo di avere il seguente codice:

```
    lea esi, [eax+4]
    les edi, [eax+2c]
    mov ecx, 4
    rep movsb
    .....
@1  ror esi-8, 3
    ...
@2  shr es:edi-7,2
    .....
    lea esi, [eax+4]
    les edi, [eax+2c]
    mov ecx, 4
    rep cmsb
```

Le prime 4 istruzioni (come avrete certamente capito) sono quelle classiche per effettuare una copia di una stringa in un'altra. Supponiamo che l'algoritmo successivamente esegua delle modifiche nelle due stringhe e poi le confronti di nuovo e che noi vogliamo capire come le modifica. Per rendere più semplice comprendere cosa il programma stia facendo possiamo usare la DATA WINDOW, ma abbiamo bisogno di 2 finestre! No problem, digitiamo DATA 0;D ESI;DATA 1;D ES:EDI; dopo le aver eseguite le due istruzioni lea e les, quindi quando ci servirà vedere cosa succede alla prima stringa (ad esempio nella locazione "@1"), digiteremo DATA 0, successivamente se dovremo dare uno sguardo alla seconda stringa (ad esempio nella locazione "@2") digiteremo DATA 1 e così via.

Altro esempio:

```
mov eax, offset stringa
push eax
call PaulaAbdul
jmp fine
```

```
PaulaAbdul: push ebp
mov ebp,esp
.....
ret
```

Supponiamo (che palle suppongo sempre!) di avere una stringa, e di metterci in eax il suo indirizzo quindi di passarlo come parametro ad una procedura (PaulaAbdul... che figa!) che la modifichi in qualche modo. Noi ci troviamo all'inizio della procedura e vogliamo vedere che cosa essa fa' alla stringa, il problema è che se digito D ESP+4 non vedrò la stringa ma l'indirizzo della stessa, e allora come fare? Beh **SoftICE** è troppo grande e ci viene in aiuto; basta saperlo usare :P La sintassi delle espressioni di **SoftICE** è simile a quella del C, quindi un programmatore C saprà già come scrivere, per chi non conosce il C ecco come fare: digitate D *(esp+4) oppure D esp->4 e vedrete magicamente la vostra stringa comparire sul video! Naturalmente potete usare anche EBP per visualizzare i parametri (l'indirizzo della stringa in questo caso), la qual cosa può esservi utile per visualizzarli anche nel mezzo della routine (cioè dopo che sono state eseguite altre istruzioni push e/o pop), per fare questo però dovreste utilizzare D *(EBP+8). Nota che *(ebp+8) deve essere utilizzato solo dopo l'esecuzione del prologo della procedura (le prime 2 istruzioni nell'esempio).

Ancora un esempio:

```
mov edi, buffer
@1: mov esi, src
mov ecx,5
rep movsb
mov edi, buffer2
@2: mov esi,src2
mov ecx,5
rep movsb
```

Questo pezzo di codice esegue delle copie di stringhe, supponiamo di voler vedere cosa faccia. Vogliamo ora vedere cosa copia in buffer ed in buffer2; per far questo possiamo procedere in due modi. Primo: eseguiamo il comando D edi quando ci troviamo all'istruzione in @1 vediamo cosa copia dopo la rep movsb e poi rieseguiamo di nuovo il comando D edi quando ci troviamo all'istruzione in @2, quindi controlliamo di nuovo cosa copia. Secondo: usiamo il comando DEX 0 edi-5, cioè controlliamo il valore di edi-5 passo passo. Così facendo dopo la prima rep movsb possiamo vedere il risultato della copia, e questo accade anche per la seconda rep movsb. Il problema che mentre tracciamo il programma non abbiamo informazioni valide sulla finestra dati perché ci viene visualizzata edi-5

e non edi; in realtà ogni volta che è cambiato edi ne vedremo le conseguenze nella finestra dati. Il perché ho dovuto aggiungere -5 ad edi è semplice, siccome edi nella rep movsb è incrementato se avessimo usato edi non avremmo visto niente, con il -5 a fine rep movsb edi si trova nell'ultimo carattere della stringa e sottraendoci 5 la possiamo vedere tutta. Il primo metodo è senz'altro preferibile al secondo, ma penso che questo esempio sia stato chiaro per capire la differenza tra D e DEX.

Ok,Ok, un'ultimo semplice esempio e poi via con una nuova sezione ;)

```
    call faiqualkosa
@1: test eax,eax ;esegue un and logico tra gli operandi e modifica solo i flags
    jnz oltre
    ....
    call qualkosaltra
    ...
    jump DaUnAltraParte
    ....
oltre: push eax
      push ebx
      call CreaIlMondo
      .....
      jmp SuPerLeMontagne
```

Hehe, trascurando le mie bellissime label, supponiamo di trovarci dopo la prima call (@1) con eax impostato a 3, e il nostro scopo è saltare ad "oltre". Possiamo usare due metodi, il primo è quello di saltare direttamente ad oltre ed il secondo è di cambiare il contenuto di eax con 0. Il primo metodo è molto semplice, basta infatti digitare REIP OLTRE (ricordo che "oltre" è una label quindi al tempo di esecuzione è un indirizzo, un numero!), cioè cambiamo il valore del registro EIP con OLTRE, in modo che la prossima istruzione sia proprio "push eax". Per quanto riguarda l'altro metodo basta digitare REAX 0, cioè cambiamo il registro EAX con 0 di modo che sia eseguito il salto ad oltre (l'istruzione test eax,eax con eax impostato a zero, non potrà che dare come risultato 0, quindi il flag Zero sarà abilitato ed il successivo salto eseguito) o in alternativa si può utilizzare il comando R FL Z dopo l'istruzione test, in modo tale da attivare il flag Zero. A prima vista i due metodi possono sembrare equivalenti, ma dando un'occhiata approfondita al codice ci accorgiamo che non è così. Il problema è che la procedura CreaIlMondo ha come parametro il risultato della prima call (cioè eax), questo non è un problema se è stato eseguito il secondo metodo per saltare ad oltre, ma lo è se invece avete usato il primo. Infatti, con il primo metodo avete saltato ad oltre con un valore di eax pari ad 3, tal valore sarà passato come parametro per la procedura CreaIlMondo, come conseguenza di ciò possiamo andare incontro ad una situazione non prevista dal programmatore, dato che il normale flusso del programma impone che eax sia 0, oppure ad un'ulteriore check del valore eax. Quindi il secondo metodo è senz'altro preferibile al primo perché ci permette di evitare possibili bug dovuti al flusso del programma non corretto.

2.3 Comandi di gestione dei Breakpoint

In questa sezione presenterò i comandi per la gestione dei BreakPoint, un'altra serie di comandi fondamentali per il **SoftICE**. Per BreakPoint si intende un "punto" in cui l'esecuzione è bloccata ed il controllo è passato al debugger. Il cosiddetto "punto" non è altro che un indirizzo di memoria. Questa definizione è valida in generale ma come vedremo, ci sono anche altri tipi di BreakPoint.

BPX Address [IF espressione] [DO Sice-Action]

Setta un breakpoint all'indirizzo "Address". L'esecuzione del programma è fermata all'istruzione di indirizzo "Address" se la condizione in "espressione" è verificata, quindi l'azione "Sice-Action" sarà eseguita. Consultare il paragrafo " **SoftICE** Avanzato" per una descrizione e qualche esempio sulle

azione
e le espressioni.

BPM Address [R|W|RW|X][Debug Register][IF espressione] [DO Sice-Action]
Setta un breakpoint di lettura/scrittura in memoria nella locazione specificata da "Address". Cioè l'esecuzione è fermata in caso sia letto e/o scritto in "Address". "Address" è un indirizzo lineare, "Debug Register" è un registro di debugging (DR0,DR1,DR2,DR3) da cambiare solamente se l'applicazione ne usa uno in particolare (in questo caso quel particolare registro non si può usare come "Debug Register"). Il parametro:

"R" indica che **SoftICE** deve fermare l'esecuzione nel caso di lettura nell'indirizzo "Address" "W" indica che **SoftICE** deve fermare l'esecuzione nel caso di scritture nell'indirizzo "Address"

"RW" indica che **SoftICE** deve fermare l'esecuzione nel caso di lettura/scrittura nell'indirizzo "Address"

"X" indica che **SoftICE** deve fermare l'esecuzione nel caso sia eseguita un'istruzione all'indirizzo "Address".

In tutti i casi tranne che nel caso in cui sia specificato "X" come parametro l'esecuzione è bloccata subito dopo l'istruzione che ha tentato di leggere/scrivere in "Address". In aggiunta a questo comando si possono usare i suffissi B,W,D per indicare la dimensione della parola da controllare. BPMB indica che la variabile in "Address" sia un byte, BPMW indica che "Address" sia una word, BPMD indica che "Address" sia una doubleword. Le opzioni di default sono, RW e il byte come suffisso, per quando riguarda il debug register ne è utilizzato uno libero a partire da DR3. Consultare il paragrafo " **SoftICE** Avanzato" per una descrizione e qualche esempio sulle azioni e le espressioni.

BPR StartAddress EndAddress [R|W|RW|T|TW] [IF espressione] [DO Sice-Action]
Setta un breakpoint di lettura/scrittura in un blocco di memoria di compreso tra "StartAddress" ed "EndAddress". "StartAddress" e "EndAddress" sono due indirizzi, "R", "W", "RW", hanno lo stesso significato già descritto per il comando BPM, "T" e "TW" saranno spiegati nella sezione **SoftICE** avanzato. BPR usato con i parametri "R", "W" o "RW" funziona come BPM con la sola differenza che in questo caso la dimensione dell'area di memoria che **SoftICE** deve controllare è definibile e non di grandezza fissa come nel caso di BPM. Nota bene in alcune versioni di **SoftICE** BPR non blocca l'esecuzione di codice a ring 0 in win9x. Nota: non esiste nella versione di Sice per NT.

BPRW ModuleName|Selector [R|W|RW|T|TW] [IF espressione] [DO Sice-Action]
E' uguale al comando BPR con la sola modifica che invece di indicargli come parametri gli indirizzi iniziali e finali del blocco da controllare, basta passargli il nome del modulo da debuggare oppure un selettore. "ModuleName" indica il nome di un modulo attivo in memoria, "Selector" indica un selettore. Anche in questo caso in alcune versioni di Sice non funziona nel caso di codice a ring 0 in Win9x. Nota: non esiste nella versione di Sice per NT.

BPIO [-h] port [R|W|RW] [IF espressione] [DO Sice-Action]
Setta un breakpoint in un'operazione di Input/Output su una porta specificata. "port" indica la porta da controllare, "R", "W", "RW", hanno lo stesso significato già descritto per il comando BPM, e l'opzione "-h" è usato solo in win9x per permettere a Sice di bloccare anche codice a ring 0.

BPINT IntNumber [IF espressione] [DO Sice-Action]
Setta un breakpoint su un determinate interrupt. "IntNumber" indica il vettore d'interrupt da controllare. L'esecuzione è quindi bloccata se è invocato l'interrupt numero "IntNumber".

BMSG WindowHandle [L] [BeginMsg [EndMsg]] [IF espressione] [DO Sice-Action]
Setta un breakpoint su un messaggio di Windows o su un range di messaggi Windows. "WindowHandle" è un handle di una finestra, "BeginMsg" e "EndMsg" indicano il range dei messaggi da controllare, "L" indica che il messaggio deve essere visualizzato sulla finestra comandi. Siccome i messaggi di Windows non sono altro che dei numeri interi, indicare 2 messaggi in "BeginMsg" e "EndMsg" significa che tutti i messaggi il cui numero è compreso tra "BeginMsg" e "EndMsg" (intesi come numeri interi) causeranno un breakpoint.

BL

Visualizza la lista e il numero dei breakpoint che sono stati settati.

BC lista|*

Cancella uno o più breakpoint precedentemente settati. "lista" è un serie di numeri separati da uno spazio o dalla virgola che rappresentano i numeri dei breakpoint da cancellare. "*" indica invece tutti i breakpoint devono essere cancellati.

BD lista|*

Disabilita temporaneamente uno o più breakpoint precedentemente settati. "lista" è un serie di numeri separati da uno spazio o dalla virgola che rappresentano i numeri dei breakpoint da disabilitare. "*" indica invece che tutti i breakpoint devono essere disabilitati. Un breakpoint disabilitato può essere di nuovo attivato tramite il comando BE.

BE lista|*

Abilita uno o più breakpoint precedentemente settati. "lista" è un serie di numeri separati da uno spazio o dalla virgola che rappresentano i numeri dei breakpoint da abilitare. "*" indica invece che tutti i breakpoint devono essere abilitati. Per disabilitare temporaneamente uno o più breakpoint usare il comando BD.

BH

Mostra tutti i breakpoint settati (anche quelli eliminati) durante la sessione corrente e ne permette di selezionarne alcuni per poterli settare di nuovo (nel caso gli avete cancellati).

BSTAT [BreakNumber]

Visualizza una statistica sul breakpoint indicato dal numero "BreakNumber". Senza parametri visualizza un statistica generale su tutti breakpoint.

BPE BreakNumber

Permette di modificare il breakpoint numero "BreakNumber".

BPT BreakNumber

Carica nell'editor la linea di comando del breakpoint numero "BreakNumber". Permette quindi di poter facilmente creare un nuovo breakpoint simile ad uno precedente.

CONSIDERAZIONI:

Ora che ne dite di qualche esempio? Ok, supponiamo di avere questo frammento di codice:

```
mov eax,0
Push eax
Push offset Titolo
Push offset Messaggio
Push eax
Call MessageBoxA
```

```
@1: ror ecx,30
....
```

questo frammento come avrete capito non fa' altro che visualizzare una message box. Ora se vogliamo bloccare l'esecuzione dell'appz nel momento in cui essa visualizza la message box, la cosa da fare è settare un breakpoint sulla API MessageBoxA usando il comando BPX MESSAGEBOXA. Una volta settato il Breakpoint e rieseguito il frammento di codice, vedrete che il programma si blocca prima di visualizzare il message box e che Sice ci mostra il codice della MessageBoxA. Più precisamente ci troviamo nel codice della dll di sistema user32, per tornare al programma chiamante e precisamente all'istruzione ror (quella dopo la call), si possono utilizzare alternativamente i tasti F11 o F12. F11 coincide con il comando G @SS:ESP (cioè il programma si fermerà giusto nell'istruzione di indirizzo @1, il quale si trova sullo stack in SS:ESP dopo aver eseguito la call) mentre F12 corrisponde al comando P RET; come già detto entrambi fanno la stessa cosa, ma con una differenza: se incominciate il trace della dll user32 potreste incontrare qualche istruzione che modifichi lo stack (quindi l'indirizzo @1 non si troverà più in SS:ESP), per questo F11 dovrebbe essere utilizzato solo se non si vuole tracciare ulteriormente user32 mentre F12 può essere utilizzato dovunque nel codice. Lo stesso metodo usato per settare un breakpoint sul message box può essere usato per qualsiasi altra API e più in generale in ogni altra procedura di cui si conosca il nome (che lo si abbia caricato da una symbol table o che sia stata caricata come funzione esportata non ha importanza).

Altro esempio:

```
mov eax, offset strA
mov ebx, offset strB
push eax
push ebx
call HoVintoAlSuperEnalotto
....
```

HoVintoAlSuperEnalotto:

```
mov eax,[esi+8]
mov ebx,[esi+4]
mov esi,eax
mov edi,ebx
mov ecx, 3
@1: rep movsb
mov ecx,[ebx]
ror ecx,3
...
@2: mov [ebx],34
Call MiSonoCompratoUnaFerrari
ret 8
```

Questo esempio può essere utile per capire come funzionano i breakpoint in memoria. Supponiamo che strA e strB siano due stringhe e che la call "HoVintoAlSuperEnalotto" le manipoli, ad esempio per produrre una chiave, supponendo che il codice sia molto complesso, noi non sappiamo dove queste siano modificate. E' questa la situazione ideale per usare una breakpoint in lettura/scrittura, infatti possiamo settare un breakpoint sulla variabile strB con il comando BPM strB W o BPR strB strB+3 W (supponendo che 3 sia la lunghezza max della stringa). Quindi una volta eseguito il programma, esso si fermerà ogni volta che si tenterà di scrivere alla locazione strB, e più precisamente si fermerà alle istruzioni di label @1, e @2. Se vogliamo fermare anche in caso di lettura della stringa bastata modificare il comando con BPM strB RW. Ricordarsi che il comando BPM può essere usato con i suffissi B,W,D nel caso che vogliate che il programma setti un breakpoint in una locazione/variabile che sia di tipo rispettivamente Byte, Word, Doubleword. Notate che in questo esempio ho utilizzato i comandi BPM e BPR indifferentemente, ma questo in realtà non è

esatto, infatti se è cambiato solo il carattere strB+2, se abbiamo settato il breakpoint con BPM Sice non bloccherà l'esecuzione mentre con il BPR Sice farà il suo dovere. Questo perché BPM strB, indica a Sice di monitorare solo il byte strB e non il byte strB+2 (ad esempio) mentre BPR strB strB+3 indica di monitorare i byte compresi tra strB e strB+3 e quindi anche strB+2. Ora avrete capito come si usano i breakpoint, e soprattutto quando siano importanti ma voglio lo stesso farvi un esempio di un altro tipo di breakpoint molto utile: il breakpoint su un messaggio Windows. Supponiamo di avere una finestra con un bottone e supponiamo di voler bloccare l'esecuzione dopo che il messaggio wm_command è ricevuto dall'appz (cioè dopo che è stato premuto il bottone!). Per farlo dobbiamo usare il comando HWND (restituisce un handle di una finestra, comunque sarà spiegato nel prossimo paragrafo) e cercare l'handle della finestra che contiene il bottone, quindi ora basta settare il breakpoint in questo modo BMSG handle wm_command, e una volta cliccato il bottone l'appz si fermerà dove voluto.

Gli altri comandi spiegati in questa sezione sono quelli relativi alla gestione dei breakpoint stessi. Ogni breakpoint settato in Sice è rappresentato da un numero, per vedere la lista dei breakpoint attuali basta usare il comando BL. Una volta conosciuto il numero corrispondente al breakpoint, si possono ora disabilitare, abilitare, eliminare, modificare etc, chiamato il comando appropriato con questo numero. Ad esempio se digitiamo BL, ci comparirà una lista di questo tipo:

```
00) BPX #0167:400000
01) BPMB #0167:402004
03) BPX USER32!MessageBoxA
```

il primo numero in questa lista indica il num. identificativo poi c'è il tipo di breakpoint e dove è stato settato. Se ad esempio vogliamo disabilitare il breakpoint sull'API MessageBoxA (la numero 3), possiamo usare il comando BD 3. Se le vogliamo disabilitare tutte usiamo il comando BD *, se invece vogliamo riabilitarle usiamo BE *. Possiamo anche ad esempio modificare il breakpoint 3 usando BPE 3, oppure creare un breakpoint simile ad esempio al numero 1, usando BPT 1, e così via tutti gli altri comandi di gestione.

2.4 Comandi per la gestione delle finestre di Sice

WC [WindowSize]

Abilita/disabilita la finestra di codice se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize" righe.

WD [WindowSize]

Abilita/disabilita la finestra dati se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize" righe.

WR

Abilita/disabilita la finestra dei registri.

EC

Muove il cursore sulla finestra di codice o dalla finestra di codice alla finestra comandi. Se la finestra di codice non è visibile la visualizza.

WW [WindowSize]

Abilita/disabilita la finestra Watch se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize" righe. La finestra Watch sarà spiegata nella sezione AVANZATA.

WL [WindowSize]

Abilita/disabilita la finestra delle variabili locali se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize"

righe. La finestra delle variabili locali v'erra spiegata nella sezione AVANZATA.

WF [-d] [b | w | d | f | *]

Visualizza la finestra dei registri della FPU o del MMX. Se non è stato specificato nessun parametro visualizza la finestra dei registri FPU. "-d" indica a Sice di visualizzare i registri, la status word e la control word dell'fpu nella finestra comandi. Gli altri parametri, cioè "b", "w", "d", "f" indicano rispettivamente di visualizzare lo stack dell'FPU in Byte, Word, DoubleWord o Real da 10 byte; nel caso della visualizzazione di tipo Real i registri saranno visualizzati come ST0-ST7, negli altri casi invece saranno visualizzati come MM0-MM7. L'ultimo parametro "*" cambia ciclicamente il tipo di visualizzazione attuale da byte->word->doubleword->real->byte etc. Siccome i registri dell'FPU sono utilizzati anche dalle istruzioni MMX è necessario scegliere il tipo di visualizzazione attuale, la visualizzazione di tipo MMX o FPU. I registri saranno chiamati MM0-MM7 per i registri MMX, e ST0-ST7 per i registri FPU, naturalmente sta' all'utente capire se quei dati saranno utilizzati da istruzioni MMX o FPU (basta vedere il codice che gli utilizza!).

CONSIDERAZIONI:

Beh dai, degli esempi sull'utilizzo delle finestre sono un'offesa al lettore, quindi non li faccio. :D

2.5 Altri comandi utili

Presenterò qui una serie di comandi utili per l'utilizzo generico di [SoftICE](#).

PEEK AddrFis

Legge un byte da un indirizzo fisico specificato "AddrFis". Si possono usare i suffissi B,W,D per leggere rispettivamente un byte, una word o una doubleword. Come il manuale di Sice stesso indica, questo comando è utile se vogliamo leggere qualche registro mappato in memoria oppure se stiamo scrivendo un programma a ring 0. Nota bene "AddrFis" è un indirizzo fisico non lineare!

POKE AddrFis

Scrive un byte in un indirizzo fisico specificato "AddrFis". Si possono usare i suffissi B,W,D per scrivere rispettivamente un byte, una word o una doubleword. Nota bene "AddrFis" è un indirizzo fisico non lineare!

FAULTS [ON|OFF]

Abilita o disabilita la caratteristica di Sice Fault Trapping. Cioè se il Fault Trapping è on, ogni volta che c'è un'eccezione del processore il controllo tornerà a Sice, il quale visualizzerà la sezione di codice dove è stato commesso l'errore.

I1HERE [ON|OFF]

Abilita o disabilita la caratteristica di [SoftICE](#) di trappare l'INT 1. Nel caso I1HERE è on, ogni volta che il processore tenta di eseguire l'interrupt 1 il controllo passa a Sice. Questo è utile per il debug dei programmi, basta infatti inserire un'istruzione INT 1 per permettere a Sice di debuggare il programma in quel punto.

I3HERE [ON|OFF|drv]

Abilita o disabilita la caratteristica di [SoftICE](#) di trappare l'INT 3. E' simile a I1here solo che questa volta l'interrupt da trappare è il 3, questo oltre ad essere utili per il debug dei propri programmi, può essere utile per la comunicazione tra Sice ed un altro debugger.

ZAP

Rimpiazza un'istruzione INT 1 o INT 3 con NOP. L'istruzione è sostituita con NOP solo se essa è l'istruzione precedente al corrente CS:EIP (o CS:IP). E' utile nel caso avete messo qualche INT 1 o INT 3 nel codice per debuggarlo, ma ora non volete più che **SoftICE** si blocchi in quella posizione.

GENINT INT1 | INT3 | NMI | IntNumber

Genera un'interrupt. I parametri possibile solo "INT1" o "INT3" per generare gli omonimi interrupt, oppure "NMI" per generare un'interrupt non mascherabile, infine un "IntNumber", dove "IntNumber" rappresenta il numero dell'interrupt da generare.

PAUSE [ON|OFF]

Abilita o disabilita lo scroll-mode. Con scroll-mode è on, se l'output di un comando Sice non può essere visualizzato tutto nella finestra comandi, esso è visualizzato solo una parte e vi è chiesto di premere un tasto per visualizzare la parte successiva.

CODE [ON|OFF]

Abilita o disabilita la visualizzazione dei byte corrispondenti alle istruzione assembler. Se CODE è on, nella finestra di codice sono visualizzate oltre alle istruzioni i byte in formato esadecimale che le rappresentano.

LINES [25|43|50|60]

Modifica il numero di linee visualizzabili sullo schermo.

FLASH [ON|OFF]

Abilita o disabilita il refresh dello schermo durante i comandi P o T.

CLS

Cancella la finestra comandi.

RS

Visualizza lo schermo del programma che si sta debuggando; premere un tasto per tornare in Sice.

ALTSCR on|off

Ridire l'output di Sice su un altro monitor.

WHAT [name | expression]

Identifica il tipo di un'espressione o una variabile e ne visualizza il risultato nella finestra comandi.

HWND [-x] [hwnd]

Visualizza le informazioni sulle handle delle finestre di Windows. La struttura di questo comando cambia a seconda che si utilizzi Sice per Nt o win9x, per questo ho indicato solo i parametri principali comuni ad entrambe le versioni di Sice. "hwnd" è un handle di una finestra, Sice visualizza le info su questa e su tutte le child. "-x" indica a Sice di visualizzare un maggior numero di informazioni sulle finestre.

EXP [[modulo!][NomeParziale*]] | [!]

Visualizza le export caricate. "NomeParziale" indica la parte di un nome di una funzione esportata da una dll o un eseguibile, da visualizzare; l'* è importante perché indica di cercare tutti i nomi di funzioni che terminano con qualsiasi combinazione di carattere. "modulo" indica un modulo in cui visualizzare le export che iniziano per "NomeParziale*". "!" indica invece di visualizzare tutti

i moduli di cui Sice ha caricato l'export. Nota in alcune versioni di Sice non è necessario aggiungere l'* per cercare le export che iniziano con quel nome.

SYM [SectionName! |!] [SymbolName [Value]]

Visualizza le info sui simboli di una symbol table caricata o setta una locazione per un simbolo. "SectionName" è il nome di una sezione nella tabella dei simboli, se è specificato solo "SectionName!", Sice visualizza tutti i simboli nella sezione "SectionName". Se è specificato solo "!", Sice mostra tutte le sezione nella symbol table corrente. "SymbolName" è il nome di un simbolo, si può anche usare un nome parziale terminato da un'* per selezionare più simboli. Se è specificato solo "SymbolName" senza "Value" sono visualizzate le informazioni su il/i simbolo/i scelti. "Value" è un indirizzo relativo, l'indirizzo assoluto sarà base della sezione del simbolo da cambiare + "Value". Se "Value" è specificato, il simbolo "SymbolName" cambierà indirizzo e gli sarà assegnato l'indirizzo base della sezione + "Value". Per un approfondimento sulle symbol table vedere la sezione AVANZATA.

SYMLOC [segment-address | o | r | -c process-type | (section-number selector linear-address)]

Riloca una symbol table. "segment-address" specifica il segmento su cui rilocare la symbol table, è usato nelle applicazioni DOS. "o" e "r" sono usati nelle appz win 16 consultare il manuale di Sice per un approfondimento su questi parametri. "-c process-type" specifica un nuovo contesto per la table, è usato nel debugging di appz DOS EXTENDER (ad es. dos4gw). Infine i parametri più usati: "section-number" indica il numero della sezione da rilocare; si può rilocare una sola sezione all volta per table win 32. "selector" il selettore da usare nella rilocazione. "linear-address" è il nuovo l'indirizzo base per la sezione che stiamo rilocando.

Una volta usati questi tre parametri la nostra sezione nella symbol table corrente avrà come indirizzo base "selector":"linear-address". Per un approfondimento sulle symbol table vedere la sezione AVANZATA.

SRC

Modifica la visualizzazione della finestra di codice tra: i sorgenti dell'appz, src + codice asm, solo codice asm.

TABLE [partial-table-name] | autoon | autooff | NomeTable

Visualizza le tabelle dei simboli caricate se non sono stati specificati parametri. "partial-table-name" è il nome parziale di una symbol table deve essere univoco (cioè lungo abbastanza da identificare un'unica symbol table), "NomeTable" invece indica il nome completo di una symbol table; se uno di questi due parametri è specificato allora la tabella corrente sarà cambiata con quella specifica nel parametro. "autoon" e "autooff" abilitano/disabilitano l'auto table switching mode, cioè la capacità di Sice di cambiare automaticamente la symbol table corrente a seconda del target. Se è specificato "autooff" l'appz corrente verrà debuggata con la symbol table corrente.

FILE [[*]NomeFile]

Visualizza o cambia il corrente file sorgente nella finestra di codice. Se "NomeFile" è specificato è visualizzato il file "NomeFile". Se è specificato "*" allora è visualizzata la lista dei file sorgenti nella symbol table corrente.

SS [NumLinea] [stringa]

Cerca una stringa nel file sorgente. "stringa" è la stringa da cercare. "NumLinea" è un numero di linea del file sorgente attualmente visibile nella finestra di codice, se "NumLinea" è specificato la ricerca parte da "NumLinea" altrimenti parte dalla prima riga visualizzata nella code window. Se la stringa è trovata, è allora visualizzata nella finestra di codice.

TYPES [NomeTipo]

Visualizza tutti i tipi disponibili o se specificato "NomeTipo" la definizione di quest'ultimo. "NomeTipo" è il nome di un tipo definito nel sorgente.

LOCALS

Visualizza le variabili locali relative al corrente contesto nella finestra comandi. Vedere la sezione AVANZATA.

CONSIDERAZIONI:

Questa volta invece di qualche esempio voglio spiegare qualche argomento correlato con i comandi appena descritti, cosa che può essere utile per i newbies. Per una trattazione specifica sugli argomenti rimando comunque a testi o documenti più specialistici, qui presenterò solo le nozioni basilari senza entrare nei dettagli.

Iniziamo con la differenza tra un indirizzo fisico ed un lineare. Non volendo scendere molto nei particolari possiamo brevemente dire che, l'indirizzo fisico è l'indirizzo reale in cui si trovano i dati nella RAM, mentre l'indirizzo lineare è un indirizzo virtuale in cui si trovano i dati, esso in generale non è uguale a quello fisico. Gli indirizzi usati nei linguaggi di programmazione in ambienti win32 sono indirizzi lineari, gli indirizzi fisici di solito sono utilizzati solo nei driver di periferica o in altre appz molto dipendenti dalla macchina.

Altro argomento di cui intendo parlare è il trapping, cioè la possibilità di Sice di attivarsi al verificarsi di un particolare evento. Dei trap sugli INT 1 o 3, già ne ho parlato, voglio ora invece parlare del trapping delle eccezioni del processore. Specifico eccezioni del processore perché queste possono essere diverse da quelle messe a disposizione di linguaggi ad alto livello come il Delphi, i quali utilizzano una combinazione di eccezioni del processore ed alcune virtuali specifiche della struttura stessa del linguaggio. Un esempio può essere più chiaro di mille parole; nel delphi ci sono delle eccezioni che sono causate ad esempio dai controlli stessi, ad esempio se si prova a leggere la linea 10 di una listbox che ha solo 5 linee ci sarà un'eccezione virtuale in quando è il controllo stesso che si accorge dell'errore nei parametri e la genera. Se invece sempre in delphi cerchiamo di leggere una locazione di memoria che non è indirizzabile questo procurerà un'eccezione che questa volta sarà hardware perché è il processore stesso che la genera non potendo leggere in quell'indirizzo. Ho fatto l'esempio del delphi ma questo può essere esteso anche ad altri linguaggi ad alto livello. Fissata cosa sia un'eccezione del processore, continuiamo a parlare di Sice. Sice, quindi, ha la possibilità di trappare le eccezioni, questo ci permette di vedere il codice che l'ha generata ed eventualmente di risolvere l'errore. Cosa questa, che può essere utile sia in fase di debugging di una propria applicazione, sia nel caso ci sia un errore in un'applicazione altrui e noi vogliamo capirne il perché. Un ottimo esercizio è infatti quello di provare a risolvere gli errori causati da un'applicazione, cosa che da un lato permette di conoscere più a fondo Windows e da un altro vi permetterà ad esempio di creare un patch che elimini il bug! (Uaooo la cosa farà sicuramente innervosire i veri programmatori dell'appz, mentre sarà motivo di orgoglio per il reverser!). Sinceramente a lungo andare, visto la grande propensione al crash delle applicazioni Windows, il trapping comincia a dar fastidio per questo consiglio di impostare il trapping a off all'inizializzazione e cambiarlo solo nei casi in cui veramente serva.

Altro argomento di cui dovrei parlare sono le symbol table, argomento che verrà esaurientemente (penso) trattato nella sezione AVANZATA.

Infine voglio sprecare qualche linea di testo per le export. Ogni file PE, che altro non è che un eseguibile o una dll in WIN95/NT, ha una sezione per le esportazioni delle funzioni. Cioè un'eseguibile o una dll può esportare delle funzioni che poi altri processi sono liberi di chiamare. Questo principio è alla base di Windows basti pensare che le API (Application Program Interface) altro non sono che delle funzioni esportate dalle librerie di sistema. Le export sono contraddistinte da un nome e da un numero. Il nome può essere omesso mentre il numero no. Questo permette a Sice di visualizzare i nomi delle funzioni esportate che sono state caricate.

Per permettere a Sice di utilizzare i nomi delle export di una dll o di un eseguibile bisogna prima farglieli caricare, per farlo basta eseguire il symbol loader e dal menù file scegliere il comando "LoadExport", quindi scegliere il file. Una volta caricata una export table di un PE (Portable Executable), i nomi delle funzioni esportate sono disponibili (le si può visualizzare con il comando EXP), e si possono anche settare dei breakpoint su di esse. Facciamo un semplice esempio:

supponiamo di avere una dll di nome ciao.dll che esporta la funzione Italia. Un programma che importerà la funzione Italia avrà un codice di questo tipo:

```
mov eax, PrimoPar
push eax //primo parametro
mov eax, SecondoPar
push eax //secondo parametro
call Italia
```

Ora supponiamo di caricare in Sice questo programma e che esso sia stato compilato senza le informazioni di debugging, il precedente pezzo di codice sarà visualizzato in Sice in questo modo (+ o -):

```
mov eax, 401004
push eax
mov eax, 401010
push eax
call 400020
```

Se invece ora carichiamo l'export della dll ciao, il codice visualizzato adesso sarà:

```
mov eax, 401004
push eax
mov eax, 401010
push eax
call ciao!Italia
```

Questo oltre a rendere più chiara l'interpretazione del codice assembly, vi permetterà di settare dei breakpoint o usare qualche altro comando Sice utilizzando il nome Italia, invece di una locazione. Come conclusione del discorso suggerisco di caricare tutte le esportazioni delle dll utilizzate da un'applicazione prima di provare a reversarla, questo renderà senz'altro più facile la comprensione del codice stesso.

Siamo a fine capitolo, a questo punto dovrete avere già una conoscenza base del **SoftICE** ora consiglio una piccola riflessione su quanto appreso prima di passare al prossimo.

3. **SoftICE** AVANZATO

3.1 Lavoriamo con le Symbol Table

Una symbol table è una tabella che contiene informazioni sui simboli ed i tipi usati nei programmi. Se si vuole utilizzare Sice per debuggere programmi di cui si dispone dei sorgenti, si deve imporre al compilare ed al linker di inserire nell'eseguibile le informazioni di debugging. Quando Sice carica un'eseguibile in cui sono presenti le informazioni di debug, esso crea una symbol table e la utilizza durante tutto il processo di debugging. Una delle importanti caratteristiche

di **SoftICE** è quella di poter caricare una symbol table indipendentemente dall'eseguibile e di rendere disponibili i tipi e simboli utilizzati a tutti i programmi che vengono debuggati. E' quindi possibile caricare una symbol table

generale con i tipi più importanti ed utilizzarli per il debug o il reverse engineering di qualsiasi applicazione. Per quanto mi riguarda io uso una symbol table contenente i tipi standard di Windows, il file è stato compilato in delphi quindi i tipi sono quelli definiti dal file windows.pas di delphi 4. Per crearsi una symbol table basta creare un'eseguibile con un qualsiasi linguaggio con le informazioni di debugging all'interno, poi eseguire il symbol loader di Sice e caricare l'eseguibile; quindi cliccare su translate. A questo punto un file .nms o .dbg sarà creato, questo è il file che contiene la symbol table. La prossima volta che si vuole usare quella symbol table basterà caricare il file .nms .dbg. Un'altra cosa importante da sapere su **SoftICE** è che se non è caricata una symbol

table non è possibile visualizzare un tipo complesso come una stringa (cioè è possibile vederla nella finestra dati, ma non si può usare un'espressione per visualizzarla). Una volta caricata una symbol table, è possibile utilizzare nelle espressioni tutti i tipi definiti (si può anche usare il casting). Per visualizzare una stringa invece basta utilizzare una funzione interna di **SoftICE**

WSTR (per una descrizione delle funzioni interne principali di Sice vedere il paragrafo QUALCHE FUNZIONE UTILE).WSTR, secondo come indicato dal manuale di Sice serve per visualizzare una stringa in formato wide (Unicode), ma la si può utilizzare anche per visualizzare una stringa normale; l'unico inconveniente è che la stringa deve terminare con due zeri (cosa che non ci interessa più di tanto, perché è facile capire dove finisce la stringa vedendola su schermo). Ad esempio se vogliamo visualizzare una stringa che si trova all'indirizzo 401000, basta digitare ?WSTR(401000). Ricordate di caricare una symbol table (potete scaricare la mia) prima di utilizzare la funzioneWSTR, altrimenti il risultato che otterrete sarà solo quello di visualizzare la doubleword come esadecimale, decimale, e come array di 4 caratteri. Il mio consiglio finale è quello di caricare una symbol table all'avvio di Sice, per far questo basta eseguire il symbol loader, e dal menù edit-> **SoftICE** initialization->Symbols aggiungere il file (.nms o .dbg o .sym) tramite il pulsante add. Oltre alla symbol table create

da un'eseguibile delphi, ho incluso anche una symbol table creata in visual c++ che è più piccola, e con cui si possono visualizzare le stringhe in modo semplice attraverso il casting; ad esempio possiamo visualizzare la stringa puntata da edi semplicemente scrivendo ?LPSTR(edi). E questo può essere utile anche nelle Watch come vedremo nel prossimo paragrafo.

Un'ultima nota sulle symbol table: bisogna stare attenti se carichiamo una symbol table e settiamo un breakpoint su un'api, perché molto probabilmente Sice prenderà come default il simbolo nel contesto corrente (cioè l'import del programma che è nella symbol table) e non l'export della libreria di sistema. Facciamo un esempio per chiarire:

supponiamo di avere un programma di nome FIGA in memoria insieme alla sua symbol table. Supponiamo anche che il programma importi l'api MessageBoxA e che quindi questo nome sia definito come simbolo nella symbol table. Settiamo ora un breakpoint con il comando BPX MessageBoxA e poi eseguiamo il comando BL. Molto probabilmente l'output di BL riferirà di un breakpoint su FIGA!MessageBoxA e non su User32!MessageBoxA. Ora nel debugging del programma FIGA, tutto funziona regolarmente e il breakpoint non crea problemi, ma se utilizziamo la stessa symbol table per debuggare altri programmi il breakpoint molto probabilmente non funzionerà. Per ovviare a questo problema possiamo semplicemente usare il comando BPX in questo modo: BPX User32!MessageBoxA e tutto andrà come dovrebbe.

[delphi symbol table](#)
[Vc++ symbol table](#)

3.1.1 Giochiamo con le symbol table.

Le symbol table come ho già detto sono molto importanti per poter comprendere il codice di un'appz, infatti sarà molto più facile per noi ricordare un nome come "Serial" che non ricordarsi la locazione 40345f. Come avete letto nella prima parte del documento ci sono 2 istruzioni che servono per la rilocazione di una

symbol table e per la modifica dell'"offset relativo" di un simbolo, essi sono SYMLOC e SYM. Purtroppo anche se la rilocazione funziona benissimo, la modifica dell'offset di un simbolo (almeno nella ver 3.23 di Sice da me utilizzata) non sempre v`a a buon fine. Nel senso che, i simboli vengono sì rilocati nelle locazioni giuste ma non sempre vengono visualizzati nella finestra di codice al posto degli indirizzi. Per nostra buona sorte però è sempre possibile creare un file MAP e tradurlo in un file sym di Sice (lo si può fare tramite l'utility inclusa in **SoftICE** Msym.exe). Ma scrivere un file map a mano è molto noioso!

Qui

che ci viene in aiuto IDA. IDA è un disassemblatore avanzato che riconosce il tipo

di compilatore che ha generato l'eseguibile ed aggiunge tutta quella serie di simboli standard che sono stati iniettati dal compilare stesso. Questi altro non sono che le procedure base del linguaggio utilizzato e le variabili globali definite dal linguaggio stesso. In più, IDA ci dà la possibilità di dare un nome alle locazioni, di inserire dei commenti e di fare molte altre operazioni. Tutto questo ci permette una più facile comprensione del sorgente assembler. Ma come IDA ci viene in aiuto? Beh, IDA permette di generare un file map dell'eseguibile disassemblato!!. Per portare i simboli di IDA in Sice potete usare Msym oppure Idasym. Un unico avvertimento se usate msym, create il file map da ida con il nome del modulo che state dissassemblando altrimenti Sice non eseguirà la rilocazione automatica. L'unione di Ida e **SoftICE** è una delle armi più potenti nelle mani del reverser per capire come funzionano programmi molto complessi, quindi vi spingo a non sottovalutare la possibilità offertaci dai mitici programmatori di IDA e Sice!!

3.2 La finestra WATCH

La finestra Watch è una delle funzionalità base di un debugger. Nel debug normale di un programma il suo utilizzo è molto semplice ed essenziale, mentre nel caso NON si abbiamo i sorgenti dell'appz da debuggare la sua utilità può sembrare minima. E' quest'ultimo il motivo che mi ha spinto ad inserire questo argomento nella sezione AVANZATA. Prima di tutto chiariamo cos'è la finestra Watch. La finestra Watch è una finestra che visualizza variabili e/o espressioni ed il loro valore attuale. In fase di debugging è utilissima perché permette di vedere cosa accade ad ogni variabile al momento dell'esecuzione. L'utilità marginale della finestra Watch aumenta all'aumentare della complessità del programma. Se avete compilato l'appz con le informazioni di debug, la finestra Watch serve per visualizzare le variabili, sia di tipi semplici che di tipi complessi, ma mentre i tipi semplici, Sice, li visualizza in maniera chiara e concisa i tipi complessi, Sice, non li gestisce molto bene. Questa è una delle pecche di Sice, infatti essendo esso un debugger sostanzialmente a basso livello, manca di quelle caratteristiche che hanno altri debugger specifici per un particolare

linguaggio. Difatti nel caso di debugging di programmi ad alto livello è meglio utilizzare altri debugger in congiunzione con Sice, in modo da sopperire alle sue mancanze. In particolare Sice non gestisce molto bene le istanze di un oggetto, essendo esso costituito da più oggetti o da tipi molto complessi. Si pensi che nei programmi scritti in Pascal se si fa il Watch di una semplice variabile stringa ne è visualizzato solo l'indirizzo e bisogna ricorrere al casting per visualizzarla; ma questo non è tutto. Non conoscendo il tipo string del pascal non si può nemmeno fare un casting semplice (naturalmente si può ovviare al tutto utilizzando la funzione interna WSTR). Questo è il problema principale delle Watch di Sice, perciò non essendo una caratteristica molto avanzata non l'ho nemmeno menzionata nella prima parte del testo, suggerisco di usare altri debugger in congiunzione con Sice se si ha il bisogno di monitorare molte variabili. Nell'ambito del reversing invece il discorso è diverso, infatti non avendo i sorgenti e dovendo capire come il programma fa una determinata cosa allora le Watch di Sice bastano così come sono. Prima di fare qualche esempio ecco qui i comandi Sice correlati alle Watch:

WW [WindowSize]

Abilita/disabilita la finestra Watch se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize" righe.

WATCH expression

Aggiunge una espressione o una variabile nella finestra Watch.

L'espressione/variabile sarà visualizzata nel seguente modo:

- espressione
- tipo dell'espressione
- valore corrente dell'espressione nel formato corrente

Naturalmente se il tipo non è riconosciuto sarà visualizzato il tipo Void e il contenuto della variabile come doubleword.

Questi sono i due comandi principali per la gestione delle Watch. Ora qualche semplice esempio:

Watch MagicTheGathering

visualizza la variabile magicthegathering

Watch WSTR(405600)

Visualizza la stringa nella locazione 405600

Watch Lpstr(esi)

Visualizza la stringa che si trova in esi effettuando un casting. Funziona solo se si ha in memoria una symbol table di un programma C. Nota: tracciando il programma ESI cambia e quindi cambierà anche il risultato di questa Watch, non male no?

Watch *ds:esi

Watch *ds:edi

Watch *ss:esp

Sono tre esempi di Watch molto utili nel reversing di un programma! :P

3.2.1 La finestra della variabili locali

Finito di parlare delle Watch, parliamo delle variabili locali. C'è una finestra usata appunto per la visualizzazione di queste senza doverle aggiungere alle Watch ogni volta che si entra in una procedura e levarle uscendo da essa. E' molto utile nel caso si abbia compilato l'appz con le info di debug, altrimenti non serve a niente questa finestra, in quanto non sarebbe visualizzata nessuna variabile. I comandi per la gestione delle variabili locali sono:

WL [WindowSize]

Abilita/disabilita la finestra delle variabili locali se non è stato specificato "WindowSize", altrimenti modifica la grandezza della finestra in "WindowSize" righe. La finestra è automaticamente aggiornata ogni volta che si entra od esce da un sub-routine, naturalmente nel caso non si abbiano le informazioni di debugging nell'eseguibile nulla è visualizzato. Le info saranno visualizzate nel seguente modo:

- stack offset
- tipo
- valore attuale

LOCALS

Visualizza le variabili locali della routine corrente nella finestra comandi. Le info saranno visualizzate nel seguente modo:

- stack offset
- tipo
- valore attuale

Un esempio pratico è molto semplice supponiamo di avere la seguente procedura:

```

procedure ImparareSice;
var
  x: byte;
  y: integer;
begin
  x:=random(5);
  y:=random(5100);
  writeln(x+y);
end;

```

Se proviamo a tracciare il programma, ed attiviamo la finestra delle variabili locali, appena dopo il begin saranno visualizzate le due variabili locali x e y ed i corrispettivi valori. Dopo l'end le var x e y scompariranno. Più semplice di così... :P

3.3 Macro

Ma cos'è una macro? In Sice una macro non è altro che un comando o una serie di comandi a cui è dato un nome e che possono essere richiamati tramite quel nome. E' come se estendessimo i comandi Sice aggiungendone altri, con la differenza però che questi non sono altro che una combinazione di quelli predefiniti.

SoftICE permette la definizione di due tipi di macro, le macro volatili e quelle

permanenti. Le macro volatili (o run-time macro) sono quelle che una volta definite rimangono in memoria fino al prossimo riavvio di Sice, quelle permanenti invece, sono definite una sola volta e sono ricaricate automatiche all'avvio di Sice.

Per gestire le macro volatili si usa il comando MACRO:

```

Crea o modifica una macro      MACRO NomeMacro = " command1;command2;..."
Elimina una macro              MACRO NomeMacro *
Cancella tutte le macro       MACRO *
Modifica una macro            MACRO NomeName
Visualizza tutte le macro definite  MACRO

```

Per la gestione delle macro permanenti invece si deve usare il symbol loader, e utilizzare la dialog che si trova nel menu Edit-> **SoftICE** initialization Setting->Macro Definition. Per aggiungere una macro basta premere Add e per eliminarla basta premere Remove il tutto in modo visuale ;)

Ora facciamo qualche esempio così chiariamo un po' le idee sulle macro:

```
MACRO dde = "d eax"
```

Visualizza la locazione puntata da eax nella finestra dati

```
MACRO dde *
```

Elimina la precedente macro

```
MACRO ciao = "? eax; ? ebx; ? ebx"
```

Visualizza il contenuto dei registri eax, ebx, ecx

```
MACRO Juve = "? 'Yaa'"
```

Ehmmm, non la spiego...provatela

```
MACRO GENius = "? %1 + %2"
```

visualizza la somma dei due parametri d'ingresso. Esempio GENius 1 3 restituisce 4 come risultato.

```
MACRO *
```

Cancella tutte le macro definite

Per le macro permanenti si usa la stessa sintassi solo che la definizione bisogna farla nel symbol loader. Nel prossimo paragrafo vedremo qualche esempio di macro più utile dei precedenti :)

3.4 Espressioni

Le espressioni in Sice, come ho già ripetuto più volte sono simili a quelle del C. Qui riporto una tabella con gli operatori usati in Sice ed esempi (riippato dal manuale di Sice):

Operatori di redirez.	Esempi
->	ebp->8 (selez la DWord Puntata da ebp+8)
.	eax.1C (selez la DWord Puntata da eax+1C)
*	*eax (Selez il valore della DWord Puntata da eax)
@	@eax (Selez il valore della DWord Puntata da eax)
&symbol	&symbol (Selez l'indirizzo del simbolo)

Operatori Matematici	Esempi
Unary +	+42 (Decimal)
Unary -	-42 (Decimal)
+	eax + 1
-	ebp - 4
*	ebx * 4
/	Symbol / 2
% (Modulo)	eax % 3
<< (Logical Shift Left)	bl << 1 (il Risultato è: bl shiftato a sinistra di 1)
>> (Logical Shift Right)	eax >> 2 (il Risultato è: eax shiftato a destra di 2)

Operatori sui Bits	Esempi
& (Bitwise AND)	eax & F7
(Bitwise OR)	Symbol 4
^ (Bitwise XOR)	ebx ^ 0xFF
~ (Bitwise NOT)	~dx

Operatori Logici	Esempi
! (Logical NOT)	!eax
&& (Logical AND)	eax && ebx
(Logical OR)	eax ebx
== (Compare Equality)	Symbol == 4
!= (Compare InEquality)	Symbol != al
<	eax < 7
>	bx > cx
<=	ebx <= Symbol
>=	Symbol >= Symbol

Operatori Speciali	Esempi
. (Line Number)	.123 (Value is Address of line 123 in source file)
() (Grouping Symbols)	(eax+3)*4
, (Lista Argomenti)	Function(eax,ebx)
: (Op. di Segmento)	es:ebx
Function	Parola(Simbolo)
# (Prot-Mode Selector)	#es:ebx (L'indirizzo è del tipo Selector:Offset)
\$ (Real-Mode Segment)	\$es:di (L'indirizzo è del tipo segment:offset)

Ok questi sono gli operatori che possiamo usare nelle espressioni Sice, se si vuole fare qualche prova basta usare il comando ? espressione, e vedere il risultato; ad esempio possiamo digitare ? (eax+8) e vedere che ci darà il valore di eax più 8, mentre ? *(eax+8) ci darà il valore puntato dall'indirizzo, eax

più

8. Facciamo un esempio concreto così si capisce meglio, supponiamo di avere la memoria in questo stato:

```
indirizzo valore
001           0
005           12
009           5
```

e supponiamo che `eax` sia 1; allora ? (`eax+8`) ci darà come risultato 9, mentre `*(eax+8)` oppure ? (`eax->8`) ci daranno 5, infine ? `*(eax->8)` oppure ? `***(eax+8)` ci daranno 12.

Bene ora conosciamo cosa sono le espressioni, ma che utilizzo pratico possono avere?

Le espressioni possono essere sia utilizzate in fase di debugging di un'appz sia per settare un breakpoint in modo che questo si verifichi solo quando è vera una determinata condizione. Supponiamo ad esempio di voler controllare quando un'applicazione apra un determinato file e vedere che tipo di accesso fa' sul file. Il metodo più semplice per fare questo è settare un breakpoint sull'Api `CreateFileA` e controllare i parametri con cui l'appz a chiamato l'Api. L'Api `CreateFile` è così definita:

```
HANDLE CreateFile(
    LPCTSTR lpFileName, // puntatore al nome del file
    DWORD dwDesiredAccess, // tipo di accesso (read-write)
    DWORD dwShareMode, // share mode
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // puntatore alla struttura
                                                security attributes
    DWORD dwCreationDistribution, // Specifica cosa fare se il file esiste o non
                                  esiste già
    DWORD dwFlagsAndAttributes, // attributi del file
    HANDLE hTemplateFile // handle di un file template
);
```

Per una descrizione più approfondita sull'Api consultare l'SDK di m\$. Ricordo solo che una funzione chiamata in C in questo modo:

```
CreateFile(nome,accesso,p1,p2,p3,p4,p5);
```

in assembler è tradotta così:

```
push p5
push p4
push p3
push p2
push p1
push accesso
push nome
call CreateFileA
```

questo perché è usata la convenzione di chiamata `STDCALL`, come per la maggior parte delle Api.

Ora se l'applicazione apre pochi file, basta allora eseguire il comando `BPX CreateFileA`, lanciare l'appz e quando si blocca controllare i parametri. Per controllare i parametri in questo caso possiamo usare i comandi `D ESP->4`, `D ESP->8`, per controllare rispettivamente il primo ed il secondo parametro che sono il nome del file ed il tipo di accesso voluto. Digitare però questi comandi ogni volta che è aperto un file è abbastanza scoccante, per fortuna però Sice ci viene in aiuto infatti la sintassi dei comandi di breakpoint indicano che

possiamo

eseguire un'azione appena dopo che si è verificato il breakpoint. E' quindi possibile fare in modo che i comandi D, siano chiamati da Sice direttamente, ma poiché sono 2 comandi che influiscono sulla stessa finestra, se eseguiti automaticamente sicuramente ci permetteranno di vedere solo l'ultimo. Per ovviare al problema, possiamo usare un comando D e un comando ?. Quindi il breakpoint sarà così modificato:

```
BPX CreateFileA DO "D ESP->4;? ESP->8"
```

Con il breakpoint così impostato appena è chiamata l'api CreateFileA l'appz è bloccata e è mostrato, nella finestra dati, quale file sta' per essere aperto e il tipo di accesso da effettuare.

Questa soluzione come ho già detto può essere utile nel caso l'appz apra pochi file (praticamente mai!), negli altri casi è una vera rottura entrare in Sice ogni volta che è aperto un file (e questo avviene anche se il file è aperto da un altro processo). Per ovviare a questo possiamo usare le espressioni; infatti i breakpoint possono essere attivati al verificarsi di una condizione... quindi il tutto sta' nel saper settare la condizione. Nel nostro caso vogliamo fermarci solo se è aperto il file "GENius.kgb". Prima di continuare ricordo che Sice non considera le stringhe come un tipo base, quindi non esistono operatori che lavorano con le stringhe come ad esempio l'operatore di uguaglianza; ed allora come fare? Beh, in Sice una dword la si può scrivere sia come numero che come insieme di 4 caratteri, ad esempio il numero 6f616963 si può scrivere come 'ciao' (anche se in realtà si dovrebbe scrivere 'oaic', Sice lo inverte) dove 63='c' 69='i' 61='a' 6f='o'. Possiamo usare questa caratteristica di Sice per eseguire un confronto, l'unica limitazione è che sia 4 la lunghezza della stringa. Usando allora questo metodo potremmo uguagliare la prima dword del nome file con una dword scritta nel formato discusso prima. Nel nostro caso dopo il breakpoint avremo che la stringa sarà puntata da ESP+4, cioè in ESP+4 c'è un indirizzo ed a quell'indirizzo inizia la stringa. Per visualizzare la prima dword eseguite il comando ? (esp->4)->0 subito dopo il breakpoint, vedrete quindi qualcosa del tipo "W\C" cioè la stringa inizia con "C:\W". Supponiamo che il file "GENius.kgb" si trovi nella directory di Windows, allora la stringa passata sarà quasi sicuramente "C:\Windows\GENius.kgb", e la prima dword ci dà "C:\W", ma quanti file iniziano per "C:\W"... un'infinità ed allora? Possiamo fare il test sulla dword numero 4 ("Eniu"), no? Per visualizzarla basta usare (esp->4->4), il breakpoint quindi sarà:

```
BPX CreateFileA IF (esp->4->4=='Eniu') DO "D ESP->4;? ESP->8"
```

Carino no ;P.. Allo stesso modo si può estendere il metodo controllando anche la dword 5 oltre la 4, così:

```
BPX CreateFileA IF ((esp->4->4=='Eniu') && (esp->4->5=='s.kg')) DO "D ESP->4;? ESP->8"
```

Questo comando in realtà non funziona perché è formato da più di 80 caratteri. In effetti una delle limitazioni di Sice è proprio di accettare comandi con un massimo di 80 caratteri. Per ovviare a questo problema si può ad esempio creare una macro del tipo:

```
MACRO Nasa = "D ESP->4;? ESP->8"
```

e poi cambiare la dichiarazione del breakpoint così:

```
BPX CreateFileA IF ((esp->4->4=='Eniu') && (esp->4->5=='s.kg')) DO "Nasa"
```

così facendo abbiamo diminuito il numero dei caratteri a 74 in modo da divenire un comando valido per Sice.

Un consiglio a coloro vogliono imparare bene Sice, cercate di usare le espressioni e le macro dove è possibile questo vi faciliterà moltissimo il lavoro e vi permetterà una maggiore conoscenza di Sice e del S.O.

3.5 Il BackTrace

Il Backtrace è una delle caratteristiche più interessanti di Sice per win 9x, ed anche una delle più "gravi" differenze tra le ver di **SoftICE** per win9x e Nt. Purtroppo in Nt il backtrace non è presente e perciò tutto quello che tratterò in questo paragrafo sarà riferito esclusivamente alla ver di Sice per Win9x. Il backtrace sostanzialmente non è altro che la possibilità di Sice di memorizzare delle istruzioni mentre vengono eseguite e di poterle tracciare successivamente. In particolare è possibile visualizzare tutte le istruzioni che sono state eseguite in un certo intervallo di memoria e simularne un trace su di esse, controllando così il flusso reale del programma in quell'intervallo. Il backtrace può essere utile in molti casi anche se in generale non è indispensabile. Descriviamo ora i comandi relativi al backtrace e poi facciamo qualche esempio.

`BPR StartAddress EndAddress [R|W|RW|T|TW] [IF espressione] [DO Sice-Action]`
Questo comando in parte già è stato spiegato mancava solo il riferimento ai parametri "T" e "TW" che vedremo ora. Come ho già spiegato questo comando in generale setta un breakpoint in un intervallo di memoria, ma utilizzandolo con i parametri "T" e "TW" non viene settato alcun breakpoint! In realtà con questi due parametri **SoftICE** setta sempre i breakpoint ma invece di bloccare l'esecuzione sull'istruzione che ha causato l'errore esso la memorizza nel backtrace buffer, in modo tale da poterla "rivedere" successivamente in modalità di backtrace.

I parametri "StartAddress" e "EndAddress" questa volta indicheranno l'inizio e la fine del blocco di codice in cui vogliamo che Sice memorizzi le informazioni di backtrace, tutti i jump e le call esterne a questo blocco non saranno monitorizzate e quindi non saranno visibili in modalità di backtrace. Il parametro "T" indicherà di memorizzare le informazioni di backtrace di tutte le istruzioni che saranno eseguite nel blocco definito, mentre "TW" indicherà la memorizzazione solo di quelle istruzioni che scriveranno nel blocco. Un consiglio non definite un blocco troppo grande altrimenti l'esecuzione del programma rallenterà in maniera evidente.

`BPRW ModuleName|Selector [R|W|RW|T|TW] [IF espressione] [DO Sice-Action]`
Praticamente funziona alla stessa maniera di BPR tranne per il fatto che non bisogna specificare il blocco di memoria ma il nome di un modulo o di un selettore ed il blocco sarà quindi automaticamente definito dalla size del modulo o dalla grandezza del segmento selezionato dal selettore. Praticamente il backtrace è inutilizzabile con questo comando perchè i blocchi definiti da un modulo sono molto grandi... quindi se non volete che il vostro pc si trasformi in un 8088.... ;)

`TRACE [b | off | numero]`

Trace permette di entrare, uscire dalla modalità di backtrace (backtrace simulation mode) o di visualizzarne lo stato. Senza parametri visualizza lo stato corrente della modalità di backtrace. Il parametro "off" vi farà uscire dalla modalità di backtrace. Trace seguito da "numero" invece entra nella modalità di backtrace nella posizione "numero" del backtrace buffer. Seguito invece dal parametro "b", Trace farà entrare nella modalità di backtrace a partire dall'istruzione più vecchia (la prima!) presente nel buffer.

`SHOW [B | numero] [l length]`

Visualizza le istruzioni dal buffer di backtrace. Show seguito da nessun parametro o da "b" inizia la visualizzazione dalla prima istruzione nel buffer. Seguito da "numero" invece inizia la visualizzazione dall'istruzione in posizione "numero" nel buffer. "length" indica invece il numero di istruzioni da visualizzare.

XT [R]

Esegue l'istruzione successiva a quella corrente nel backtrace buffer. Se "R" è specificato esegue invece l'istruzione precedente. Se viene incontrata un call continua dal punto d'entrata della routine solo se essa è presente nel backtrace buffer. Normalmente questo comando senza parametri è associato ai tasti "CTRL-F8", mentre con il paramentro "R" è associato ai tasti "ALT-F8".

XP

Funziona come XT ma non entra nelle call e nelle rep, ed inoltre non permette lo step all'istruzione precedente. Normalmente il comando è associato al tasto "CTRL-F10".

XG [R] indirizzo

Effettua un salto nelle istruzioni di backtrace e quindi solo nella modalità di backtrace. "indirizzo" è l'indirizzo in cui si vuole saltare, se non è trovato nel backtrace buffer viene visualizzato un messaggio di errore. Il parametro "R" permette di fare un salto all'indietro all'interno del backtrace buffer.

XRSET

Cancella tutte le informazioni contenute nel backtrace buffer, può essere usato solo quando non si è nella modalità di backtrace.

Sice solo nella modalità di backtrace permette di simulare l'esecuzione delle istruzioni registrate nel backtrace buffer, in questa modalità non è possibile eseguire la maggior parte dei comandi standard. Una delle cose fondamentali da capire sul backtrace riguarda le due diverse modalità in cui esso è implementato; esse corrispondono rispettivamente ai parametri "T" e "TW" dei comandi BPR e BPRW. Cominciamo con il parametro "T":

quando questo parametro è specificato **SoftICE** setta un breakpoint di esecuzione nel blocco di memoria definito; quando un'istruzione viene eseguita (in questo blocco di memoria) Sice ne memorizza l'indirizzo di memoria nel backtrace buffer. Quando si entra in modalità di backtrace è possibile simulare tutte le istruzioni memorizzate nel buffer e quindi controllare il flusso del programma. E' importante notare che Sice memorizza solo le locazioni delle istruzioni e non le istruzioni stesse, difatti se provate a vedere il backtrace buffer (con il comando show) in due contesti diversi (due processi diversi!) noterete che le istruzioni nel buffer sono diverse! Ciò è dovuto al fatto che l'address space di un processo è in generale diverso da quello di un'altro. Ora facciamo un'esempio concreto per capire come si può utilizzare il backtrace. Supponiamo di avere in memoria il seguente frammento di codice:

```
bingo:  mov eax,1
        mov ecx,7
        shld eax,ecx,3
        test eax,ecx
@1:    jz @2
        mov eax,0
@2:    add caz, edx
bongo:  ret
```

Ok, l'esempio è idiota come al solito ma tanto non è quello che conta! Prima di avviare l'esecuzione di questa routine eseguiamo il comando BPR CS:bingo CS:bongo T (ricordo che bingo e bongo sono due label ed a tempo di esecuzione saranno due indirizzi), poi mandiamola in esecuzione. Finita l'esecuzione della routine supponiamo di essere in Sice, ed eseguiamo il comando SHOW. Vedrete qualcosa del genere:

```
8  xxx mov eax,1
7  xxx mov ecx,7
6  xxx shld eax,ecx,3
5  xxx test eax,ecx
```

```

4 xxx jz xxx -----
3 xxx mov eax,0      |
2 xxx add caz, edx  <-|
1 xxx ret

```

Nota: al posto delle x ci sarà la locazione di memoria in cui è contenuta l'istruzione; il primo numero indica la posizione nel buffer.

Notiamo che queste altro non sono che le istruzioni che sono state eseguite nel blocco da noi definito. Poniamoci ora il problema di determinare se il salto condizionale @1 è stato eseguito o no. Per capirlo basta guardare il codice nel buffer, infatti da quando riportato si vede che il salto non è stato eseguito perchè l'istruzione successiva al salto è la mov; se fosse stato eseguito avremmo avuto un backtrace buffer di questo tipo:

```

7 xxx mov eax,1
6 xxx mov ecx,7
5 xxx shld eax,ecx,3
4 xxx test eax,ecx
3 xxx jz xxx -----
2 xxx add caz, edx  <-|
1 xxx ret

```

Certo però che se ne avessimo 300 salti invece di uno non sarebbe stato così facile controllare il flusso del programma. Come al solito ci viene in aiuto Sice che ci permette di andare in modalità di backtrace e di simulare ciò che è avvenuto quando il processore ha eseguito il frammento di codice. Per entrare in modalità di backtrace basta invocare il comando TRACE B, oppure TRACE "num" dove num indica la posizione nel buffer da cui si vuole iniziare la simulazione. Ora basta premere CTRL-F8 O CTRL-F10 per simulare i comandi X e P (F8 e F10) che si usano in condizione normale. In pratica se eseguiamo TRACE B e poi premiamo CTRL-F10 il programma simulerà l'istruzione mov eax,1 ripremendolo simulerà mov ecx,7 e così via. In questo modo possiamo controllare di persona i salti che il programma ha eseguito e quindi il flusso corretto.

Questo modo di utilizzo del backtrace può essere molto utile sia in situazioni di debugging in cui gli errori sono difficili da trovare, sia nel reverse di qualche applicazione per capire meglio quali salti ha eseguito il programma in una determinata situazione e quali in un'altra.

Il parametro "TW" indica invece un diversa modalità nella memorizzazione del buffer di backtrace. Infatti se è stato specificato tale parametro, Sice setterà un breakpoint di scrittura nel blocco da noi definito; ora quando un'istruzione scrive in questo blocco Sice memorizza la locazione dell'istruzione, con la conseguenza che il backtrace buffer in un determinato momento contiene tutte le istruzioni (e i relativi indirizzi) che hanno scritto nel nostro blocco. Noterete la sostanziale differenza rispetto al primo metodo, cioè il blocco nel primo metodo contiene codice, mentre nel secondo di solito contiene dati. Ora facciamo un esempio per esemplificare il tutto:

```

.data
SLAYER db "Diabolus in Musica",0 ;lunghezza 22 bytes
.code
bingo: mov eax,45h
        mov eax, [SLAYER]
        ror ecx,edx ; qualke istruzione a caso
        shl edx,eax
        mov eax,45 ;fine istr a caso :P
@1:    mov [SLAYER],50h
        mov edx,eax ;ancora istr a cz
        cmp edx,ecx
        ror eax,5 ;fine istr a cz

```

```
@2:    mov dword ptr [SLAYER+4],eax
bongo: mov edx, 89
```

Dando un'occhiata al solito frammento di codice ci accorgeremo che le uniche istruzioni che hanno accesso alla memoria sono la @1 e la @2. Se ora settiamo un breakpoint sull'array SLAYER, con il comando BPR DS:SLAYER DS:SLAYER+22 TW, prima di eseguire il codice, avremo come risultato finale il backtrace buffer riempito solo dalle istruzioni @1 e @2. Infatti sono solo le istruzioni @1 e @2 che scrivono nel blocco definito dal BPR, e quindi Sice memorizza solo quelle. Anche in questo caso è da notare che Sice memorizza le locazione non i byte corrispondenti alle istruzioni, per questo se si fa un context switch le istruzioni nel buffer saranno diverse. E' da notare che in questo caso la simulazione del trace non ha molto significato, in quanto le istruzioni nel buffer molto probabilmente non saranno contigue, cioè sono istruzioni che si trovano in parti di codice diverse e sono separate da altre istruzioni. Infatti se vediamo l'esempio precedente, invocando il comando SHOW vedremo una cosa del genere:

```
2 xxx  mov 403000,50
1 xxx  mov 403004, eax
```

ma tra l'istruzione mov 403000,50 e l'istruzione mov 403004,eax ci sono altre istruzioni in mezzo, quindi il trace in questo buffer non ha molto senso. In particolare non ha molto senso l'istruzione mov 403004,eax senza sapere che elaborazione viene fatta su eax prima di scriverlo nel blocco. Questo metodo di utilizzo del backtrace potrebbe servire per tener traccia delle istruzioni che scrivono in un particolare blocco e può candidarsi come utile sostituto del comando BPR con il parametro W. Altre applicazioni di questo metodo possono essere ad esempio nella ricerca di codice automodificante o il debugging di applicazioni concorrenti.

3.6 Comandi avanzati per reperire informazioni sul sistema

Alcuni di questi comandi richiedono conoscenza approfondita del Sistema Operativo per questo non sono spiegati approfonditamente, rimando ai manuali di [SoftICE](#) per un approfondimento su questi argomenti.

Addr : Visualizza o scambia l'Address Context corrente.
Class : Visualizza le informazioni sulle Windows Classes.
CPU : Visualizza i registri della CPU.
Exp : Visualizza i nomi delle funzioni esportate caricate in Sice.
GDT : Visualizza la Global Descriptor Table (GDT).
Heap : Visualizza il Global Heap di Windows.
Heap32 : Visualizza la Global 32-bit Heap di Windows.
HWND : Visualizza le informazioni sulle handles di Windows.
IDT : Visualizza la Interrupt Descriptor Table (IDT).
LDT : Visualizza la Local Descriptor Table (LDT).
LHeap : Visualizza la Local Heap di Windows.
Map32 : Visualizza la mappa di memoria di tutti i moduli a 32-bit caricati.
MapV86 : Visualizza la mappa di memoria DOS della corrente Virtual Machine.
Mod : Visualizza la lista dei Windows Module.
Page : Visualizza le informazioni sulla Page Table.
Pci : Visualizza le informazioni sul PCI.
Phys : Visualizza tutti gli indirizzi virtuali corrispondenti ad un indirizzo fisico
Proc : Visualizza le informazioni su un processo.
Stack : Visualizza il call stack.
Sym : Setta or Visualizza un symbol.
Task : Visualizza la Task List di Windows.
Thread : Visualizza le informazioni su un thread.
TSS : Visualizza il Task State Segment e gli Hooks sulle Porte di I/O.
VCall : Visualizza i nomi e gli indirizzi di una routine in un VxD.

VM : Visualizza le informazioni sulle virtual machines.
VXD : Visualizza la VXD map di Windows.
.VMM : Chiama la VMM Debug Information Services menu.
.VPICD : Chiama la VPICD Debug Information Menu.
.VXDLDLDR : Visualizza le informazioni sui Vxd.
WMSG : Visualizza i nomi ed i numeri dei messaggi di Windows.

3.7 Qualche funzione utile

In questo capitolo presenterò alcune funzioni interne di [SoftICE](#) molto utili. Alcune di queste sono solo per chi ha una conoscenza approfondita del sistema operativo.

Byte: restituisce il byte basso
--- esempio: ? Byte(0x1234) = 0x34
Word: restituisce la word bassa
--- esempio: ? Word(0x12345678) = 0x5678
Dword: restituisce la dword bassa
--- esempio: ? Dword(0xFF) = 0x000000FF
HiByte: restituisce il byte alto
--- esempio: ? HiByte(0x1234) = 0x12
HiWord: restituisce la word alta
--- esempio: ? HiWord(0x12345678) = 0x1234
Sword: Converte un byte in una word con segno
--- esempio: ? Sword(0x80) = 0xFF80
Long: Converte un byte o una word in una doubleword con segno
--- esempio: long ? Long(0xFF) = 0xFFFFFFFF
WSTR: Visualizza una string Unicode o Normale
--- esempio: ? WSTR(eax)
Flat: Converte un indirizzo relativo in formato selettore/spiazzamento in uno lineare
--- esempio: ? Flat(fs:0) = 0xFFDFF000
CFL: Carry Flag
--- esempio: ? CFL = bool-type
PFL: Parity Flag
--- esempio: ? PFL = bool-type
AFL: Auxiliary Flag
--- esempio: ? AFL = bool-type
ZFL: Zero Flag
--- esempio: ? ZFL = bool-type
SFL: Sign Flag
--- esempio: ? SFL = bool-type
OFL: Overflow Flag
--- esempio: ? OFL = bool-type
RFL: Resume Flag
--- esempio: ? RFL = bool-type
TFL: Trap Flag
--- esempio: ? TFL = bool-type
DFL: Direction Flag
--- esempio: ? DFL = bool-type
IFL: Interrupt Flag
--- esempio: ? IFL = bool-type
NTFL: Nested Task Flag
--- esempio: ? NTFL = bool-type
IOPL: livello di privilegio corrente
--- esempio: ? IOPL = livello di privilegio corrente
VMFL: Virtual Machine Flag. Nota: funziona solo su NT
--- esempio: ? VMFL = bool-type
DataAddr: Restituisce l'indirizzo del primo byte visualizzato nella finestra dati
--- esempio: dd @dataaddr
CodeAddr: Restituisce l'indirizzo della prima istruzione visualizzata nella finestra di codice

```

===- esempio: ? codeaddr
KPEB: (Kernel Process Environment Block) del processo attivo. Nota: funziona
      solo su NT
===- esempio: ? process
KTEB: (Kernel Thread Environment Block) del thread attivo. Nota: funziona solo
      su NT
===- esempio: ? thread
PID: Id del processo attivo. Nota: funziona solo su NT
===- esempio: ? pid == Test32Pid
TID: Id del thread attivo. Nota: funziona solo su NT
===- esempio: ? tid == Test32MainTid

```

3.8 Differenze tra la ver 9x e Nt

Le differenze fondamentali tra le versioni di Sice dipendono come è facilmente immaginabile dalle enormi differenze d'implementazione dei due sistemi operativi. Alcuni comandi ci sono solo in una ver e non nell'altra, anche se sostanzialmente quelli fondamentali ci sono in entrambe. Tra le differenze fondamentali c'è la NON presenza dei comandi BPR e BPRW nella ver Nt e della conseguente mancanza del backtrace. Un'altra differenza importante è la differente modalità operativa del comando TASK tra Nt e Win9x. Infatti TASK in Nt in pratica è inutile (visualizza solo le informazioni sui task a 16 bit), mentre in win 9x funziona benissimo. Consideriamo una situazione normale di una finestra con un bottone e di volerla fermare come al solito al sopraggiungere del messaggio `wm_command`. Per fare questo in win 9x usualmente si usano i seguenti comandi:

- TASK: per conoscere il nome del task da passare al comando HWND
- HWND `nometask`: per conoscere l'handle della finestra target
- BMSG `handlefinestra WM_COMMAND`

Ora questa sequenza se ripetuta in Nt non funziona, in particolare è TASK che crea problemi. Per fortuna ci sono altri comandi che si possono usare al posto di TASK, essi sono MOD e PROC. Consiglio di non usare MOD perchè ci darebbe la lista di tutti i moduli caricati, la quale come potete immaginare sarà lunghissima. Quindi potremo usare PROC, la sequenza verrà quindi così modificata:

- PROC: per conoscere il nome del processo da passare al comando HWND
- HWND `nomeprocesso`: per conoscere l'handle della finestra target
- BMSG `handlefinestra WM_COMMAND`

Notiamo che questa sequenza funziona sia nella ver di Sice per Nt che per win9x. Naturalmente consiglio di usare qualche utility per ottenere più rapidamente l'handle della finestra!

HWND tra l'altro è uno dei comandi che funziona in maniera diversa in win9x ed in Nt. Vediamone ora le differenze nei parametri tra le 2 versioni.

HWND per win9x:

```
HWND [-x][hwnd | [[level][process-name]]
```

level: rappresenta il livello gerarchico. 0 è il livello più alto, 1 il successivo. Se specificato verranno visualizzate solo le info sulle finestre del livello scelto.

hwnd: handle di finestra. Verranno visualizzate le info solo di questa finestra se questo parametro è specificato.

process-name: rappresenta un processo in memoria. Visualizza le info solo sulle finestre di questo processo.

x: Visualizza un maggior numero di informazioni sulle finestre.

In uscita il comando visualizzerà il nome della classe e l'indirizzo della window procedure.

HWND per Nt:

```
HWND [-x][-c] [ hwnd | desktop-type | process-name |  
thread-id | module | class-name]
```

x: Visualizza un maggior numero di informazioni sulle finestre.

hwnd: handle di finestra. Verranno visualizzate le info solo di questa finestra se questo parametro è specificato.

process-name: il nome di un processo. Visualizza le info solo sulle finestre di questo processo.

thread-id: l'id di un thread. Visualizza le info solo sulle finestre di questo thread.

module: indica un modulo. Visualizza le info solo sulle finestre di questo modulo.

class-name: indica il nome di una classe.

desktop-type: indica un handle ad un desktop solo per Nt 3.51.

c: forza la visualizzazione della gerarchia delle finestre quando sono stati specificati i parametri thread-id, module o class-name.

In uscita il comando visualizzerà il nome della classe, l'indirizzo della window procedure, il thread ID ed il modulo.

In effetti i parametri di HWND nelle 2 versioni sono diversi ma sostanzialmente il comando fa la stessa cosa.

Un'altra differenza, molto importante per alcuni, è che in Nt non esiste l'api hmemcpy, quindi tutti quelli che sono abituati ad usare un breakpoint a tale Api devono cambiare metodo!

Infine l'ultima differenza di cui voglio parlare è che in Nt non ci sono i selettori 30 e 28. Essi in win9x si trovano nella GDT e che rappresentano i selettori di default per il codice di VXD, hanno infatti livello di privilegio pari a 0. Per via di queste caratteristiche essi potevano essere utilizzati come selettori in particolari breakpoint in win9x, ma in Nt essi non sono più i selettori 30 e 28 (essi possono essere i selettori 9 e 10, ma non sono sicuro che essi non varino a seconda della ver di Nt installata) e quindi non possiamo utilizzarli come selettori nei nostri breakpoint.

4. QUALCHE API SU CUI SETTARE UN BREAKPOINT

Ecco una serie di API usate comunemente, su cui settare i nostri breakpoint:

OPERAZIONI SUI FILE:

```
CreateFile,CreateFileA,CreateFileW  
ReadFile  
WriteFile  
SetFilePointer
```

_lcreat
_lopen
_lread
_lwrite
_llseek

GESTIONE DELLE DIRECTORY:

GetCurrentDirectory, GetCurrentDirectoryA, GetCurrentDirectoryW
GetSystemDir *16 bit
GetSystemDirectory, GetSystemDirectoryA, GetSystemDirectoryW
GetWindowsDir *16 bit only
GetWindowsDirectory, GetWindowsDirectoryA, GetWindowsDirectoryW

OPERAZIONI SUI FILE INI:

GetPrivateProfileString, GetPrivateProfileStringA, GetPrivateProfileStringW
GetPrivateProfileInt, GetPrivateProfileIntA, GetPrivateProfileIntW
WritePrivateProfileString, WritePrivateProfileStringA,
WritePrivateProfileStringW
WritePrivateProfileInt, WritePrivateProfileIntA, WritePrivateProfileIntW

OPERAZIONI SUL REGISTRO DI CONFIGURAZIONE:

RegCreateKey, RegCreateKeyA, RegCreateKeyW
RegCreateKeyEx, RegCreateKeyExA, RegCreateKeyExW
RegQueryValue, RegQueryValueA, RegQueryValueW
RegOpenKey, RegOpenKeyA, RegOpenKeyW
RegCloseKey, RegCloseKeyA, RegCloseKeyW

OPERAZIONI SULLE FINESTRE E SULLE DIALOG BOXES:

createwindow, createwindowA, createwindowW
CreateWindowEx, CreateWindowExA, CreateWindowExW
ShowWindow
GetWindowPlacement
GetWindowLong, GetWindowLongA, GetWindowLongW
GetWindowWord
CreateDialog, CreateDialogA, CreateDialogW
CreateDialogIndirect, CreateDialogIndirectA, CreateDialogIndirectW
CreateDialogIndirectParam, CreateDialogIndirectParamA,
CreateDialogIndirectParamW
CreateDialogParam, CreateDialogParamA, CreateDialogParamW
DialogBox, DialogBoxA, DialogBoxW
DialogBoxParam, DialogBoxParamA, DialogBoxParamW
DialogBoxIndirectParam, DialogBoxIndirectParamA, DialogBoxIndirectParamW
DialogBoxIndirect, DialogBoxIndirectA, DialogBoxIndirectW
EndDialog

FUNZIONI SUI MESSAGE BOX:

MessageBox, MessageBoxA, MessageBoxW
MessageBoxEx, MessageBoxExA, MessageBoxExW
MessageBoxIndirect, MessageBoxIndirectA, MessageBoxIndirectW
MessageBeep

OPERAZIONE DI ACQUISIZIONE TESTO SU CONTROLLI DI TESTO O DI EDIT:

GetWindowText, GetWindowTextA, GetWindowTextW
GetDlgItemText, GetDlgItemTextA, GetDlgItemTextW
GetDlgItemInt

COMANDI PER VISUALIZZARE UN MESSAGGIO DI TESTO:

Textout, TextOutA, TextOutW
DrawText, DrawTextA, DrawTextW
DrawTextEx, DrawTextExA, DrawTextExW

SendMessage
wsprintf

OPERAZIONI COMUNI SUI CDROM:

GetDriveType, GetDriveTypeA, GetDriveTypeW
GetLogicalDrives, GetLogicalDrivesA, GetLogicalDrivesW
GetLogicalDriveStrings, GetLogicalDriveStringsA, GetLogicalDriveStringsW
GetVolumeInformation

OPERAZIONI COMUNI SUL TEMPO:

GetLocalTime
GetSystemTime
GetSystemTimeAsFileTime
SystemTimeToFileTime
CompareFileTime

OPERAZIONI COMUNI SULLE STRINGHE:

hmemcpy
lstrcpy, lstrcpyA, lstrcpyW
lstrcat, lstrcatA, lstrcatW
lstrcpyn, lstrcpynA, lstrcpynW
Comparestring, ComparestringA, ComparestringW

Nota: alcune API hanno 3 diversi nomi per eseguire la stessa operazione. Il primo è la versione a 16 bit della funzione, le altre due sono a 32 bit. La A o la W a fine nome denota funzioni che rispettivamente lavorano con parametri stringa di tipo normale o Wide (Unicode). Alcune vecchie applicazioni a 16 bit possono chiamare delle funzioni DOS, attraverso l'API DO3CALL oppure direttamente tramite l'istruzione assembler INT, quindi in questi casi dovrete usare un breakpoint direttamente sull'interrupt in cui volete fermarvi.

Fine