The Italian CRaCKiNG Encyclopedia vol. 2
More about CRaCKiNG

Scritto da

---==: ALoR :==---
---==> Proud member of NEURO ZONE 2 <==---

Maggio 1998

Target = ***, ****

(*=Lamer, **=Novizio, ***=Apprendista, ****=Esperto, *****=CrackMaster)

==========

Table of context :

==========

1.0 DISCLAIMER

Every reference to facts, things or persons (virtual, real or esoteric) are purely casual and involuntary. Any trademark nominated here is registered or copyright of their respective owners. The author of this manual does not assume any responsability on the content or the use of informations retriven from here; he makes no guarantee of correctness, accuracy, reliability, safety or performance. This manual is provided "AS IS" without warranty of any kind. You alone are fully responsible for determining if this page is safe for use in your environment and for everything you are doing with it!

E' in inglese lo so... ma vi avevo detto di aggionrnarvi nel primo manuale, o sbaglio.... beh, se non l'avete ancora fatto sbrigatevi... il tempo vola!!! Ok. e dopo questa menata totalmente inutile, ma che serve a me per pararmi il culo, possiamo iniziare con il secondo volume.

2.0 I TOOLS NECESSARI

I tools necessari per crackare li abbiamo già visti nel primo volume. Comunque è meglio ricordarli.

* Un disassemblatore (Es.: WDasm 8.9 or higher)
* Un editor esadecimale (Es.: UltraEdit)
* Un manuale di Assembly
* Un manuale delle API.

In oltre d'ora in poi comincerò a parlare anche di debugger in realtime, quindi è necessario che iniziate a procurarvi

* Soft-Ice 3.2 or higher (il miglior debugger per win)

Vi consiglio anche di scaricare i vari patch di soft-ice, che servono per bypassare le protezioni dei programmi che riprogrammano gli INT 3 (poi vi spiego meglio).

3.0 PENSARE COME UN CRACKER

A volte può capitare che patchando la funzione di registrazione ci venga detto che il programma è registrato, quando però lo carichiamo il programma risulta NON registrato. La soluzione è molto semplice, ma richiede che voi pensiate come un cracker.
I programmatori seguono la logica secondo la quale se una serie di istruzioni va ripetuta più di una volta, si deve creare una funzione, che si richiama ogni volta che si deve effettuare quel tipo di istruzioni. Molti programmi checkano il serial # almeno due volte, quando immetti il codice e quando il programma viene eseguito. Per questo motivo il programmatore crea una funzione, e la richiama ogni volta che checka il codice.
Se voi patchate la funzione, questa restituirà un codice valido ogni volta che questa sarà richiamata.
Facciamo un esempio. Fate attenzione alle istruzioni assembly qui riportate.

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|
|:00440A99(C)
|

* Possible Reference to String Resource ID=00141: "Unregistered"

:00440B3C 688D000000                    push 0000008D
:00440B41 8BCF                          mov ecx, edi
:00440B43 E8CD110600                    call 004A1D15
:00440B48 53                            push ebx
:00440B49 53                            push ebx

* Possible StringData Ref from Data Obj >"Registration unsuccessful". Please
                                       >"verify that you have entered the"
                                       >"information exactly as shown on"
                                       >"your registration letter."

:00440B4A 6800544F00                    push 004F5400
:00440B4F 899E78010000                  mov dword ptr [esi+00000178], ebx
:00440B55 E8CC940600                    call 004AA026
```

Avrete notato che questa parte di codice è richiamata da un salto condizionata all'indirizzo 00440A99. Scorrendo in su WDasm troveremo queste istruzioni:

```
:00440A87 50                            push eax
:00440A88 51                            push ecx
:00440A89 898678010000                  mov dword ptr [esi+00000178], eax
:00440A8F E85CF1FEFF                    call 0042FBF0
:00440A94 83C408                        add esp, 00000008
:00440A97 85C0                          test eax, eax
:00440A99 0F849D000000                  je 00440B3C
```

Ora dobbiamo applicare quello che abbiamo imparato prima. Se ci limitiamo a NOPpare il je 00440B3C o a trasformarlo in jne 00440B3C, il programma ci dirà di essere stato registrato quando immettete il codice. Quando, però, fate ripartire il programma continuerà a dirvi "Unregistered". Vi ricordate la questione delle funzioni richiamate più volte? Questo è quello che l'autore del programma ha fatto. Andiamo quindi a vedere cosa si trova all'indirizzo della call 0042FBF0.

```
* Referenced by a CALL at Addresses:

|:0040443B , :004048B8 , :004057FA , :0042E80E , :0042E886
|:00439F74 , :0043D7CA , :0043E060 , :0043ECDA , :004409F5
```

```
|:00440A8F , :0044234C , :00442CDD , :004543F9 , :004545CB
|:004BA403
|
:0042FBF0 64A100000000                  mov eax, dword ptr fs:[00000000]
```

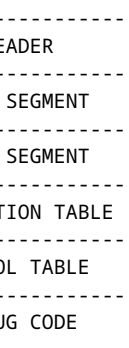
Ognuno di questi indirizzi è una chiamata alla funzione "CodiceValido()" che controlla se il programma è registrato.
"La domanda sorge spontanea..." (citazio) Come facciamo per patchare la funzione?
Diamo uno sguardo alla sezione di codice che richiama la funzione ed in particolar modo all jump verso 00440B3C. Il jump sarà effettuato se EAX = 0. Così, il modo migliore per patcharlo è scrivere 1 nel EAX. Quindi modificheremo il mov eax, dword ptr fs:[00000000] con [00000001] in modo che la funzione restituisca sempre 1. In questo modo il programma penserà di essere registrato.
Questa tecnica può essere usata non sono per i programmi con il serial #, ma anche per altri tipi di protezione (per esempio i controlli sulle date o i nag screen sono messe in una funzione).

4.0 LA TECNICA JUNGLE

I programmatori più esperti cercano in ogni modo di impedire che i loro programmi vengano crackati. Per questo inventano numerose tecniche. Una delle più semplici è la tecnica "Jungle". Viene spesso usata, poiché è di facile realizzazione, e poiché spinge il cracker ad arrendersi. Ma noi abbiamo i controcoglioni... e non ci arrenderemo... :-)
Questa tecnica, non è mirata ad impedire di crackare il programma, più che altro cerca di far desistere il cracker facendolo impazzire in una "giungla" (da qui il nome) di call e jump. Praticamente, il programma può richiamare anche 15-16 call in 4-5 DLL differenti (vedi appendice -A-) prima di arrivare al punto in cui compare la protezione.
L'unico modo per superare questo ostacolo è avere un mucchio di pazienza, un blocchetto di appunti su cui scrivere i NUMEROSI indirizzi che ci servono e magari un Martini o un pacchetto di sigarette (se fumate) :-)

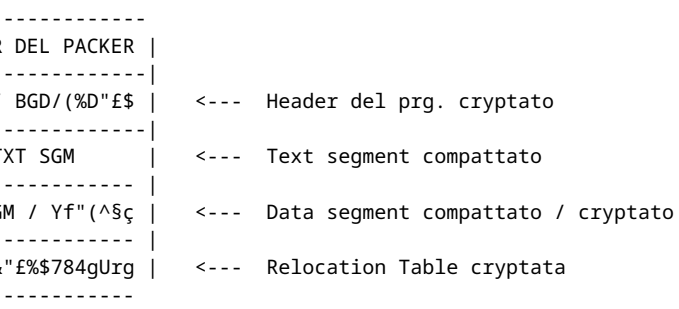
5.0 FILE PACKING/UNPACKING

Un altro modo per proteggere i programmi è il file packing/crypting.
Esistono vari programmi che compattano i file eseguibili (Pklite, WWpack, diet, ecc..). Ma possono essere facilmente decompattati per ritornare al file eseguibile originale. Le versioni commerciali di questi prg. forniscono la possibilità di renderli "unextractable", criptando il file. Esistono anche molti altri prg (non sto qui ad elencarli tutti... sono tantissimi) che cryptano i file rendendoli inestraibili, oppure indebuggabili.
Vediamo ora come funzionano e che problemi creano al cracker.
Innanzi tutto diamo una rapida occhiata a come sono organizzati i file eseguibili.



Questo è lo schema di come si presenta un file eseguibile appena compilato, e molto facile da debuggare e decompilare... :-)
Sfortunatamente per noi i packer/crypter rendono le cose più complicate.
Per prima cosa un packer elimina la "SYMBOL TABLE" e il "DEBUG CODE" che come dice il nome è pieno di informazioni utili in fase di debug. Eliminata questa

parte, che non serve per il corretto funzionamento dell'eseguibile, il packer compatta le stringhe che si trovano nel "TEXT SEGMENT". Infine se si è selezionata l'opzione per renderli inestraibili, il packer crypta l'HEADER e/o la RELOCATION TABLE.
Per poter reperire le informazioni così compattate e cryptate, il packer appende parte di codice al programma che deve essere eseguito prima di ogni altra cosa.
Dopo l'azione del packer il nostro file eseguibile si presenta così:



Il nostro file quindi si presenta come un ammasso di byte insignificanti per qualsiasi debugger. Infatti provando a disassemblare un file di questo tipo, si otterrà solo un mucchio di codici senza senso.
Vediamo allora come fa il sistema operativo ad interpretare i comandi dell' eseguibile. Quando lanciamo il programma, viene letto l'HEADER DEL PACKER e viene richiamata la routine di decrypt. Una volta decryptati i dati in memoria, la parte di codice del packer passa il controllo ai dati appena decompattati, ed esegue il vero codice dell'eseguibile.
In oltre alcuni packer inseriscono una parte di codice dedicata alla rilevazione/inibizione dei debugger.
Fortunatamente essendo algoritmi di crittazzione a due vie (crypt/decrypt), esistono programmi che decryptano questi file :-), riportandolo al suo stato naturale.
Per impedire ciò sono stati inventati altri programmi per cryptare ulteriormente gli header dei prg. Ma anche qui essendo crypt a due vie, c'è sempre il modo di tornare indietro... basta solo scoprire l'algoritmo... :-)
NON esiste il crypter SICURO !! Se per ora nessuno ha trovato la chiave, state certi che prima o poi qualcuno la troverà :-)
Ormai io ritengo che questo tipo di protezione serve solamente contro i LAMER che potrebbero, con un qualsiasi editor esadecimale, editare le stringhe del TEXT SEGMENT, e cambiare il copyright o simili...
I programmi più recenti e più BASTARDI, integrano nel codice algoritmi che rilevano i debugger o che li depistano riprogrammando gli INT (vedi 5.0).

6.0 IL PROBLEMA DEGLI INT 3 & INT 1

Il problema degli INT 3 riguarda il Tracing con i debugger in realtime.
A volte, quando inizi a crackare, il tuo "instruction pointer" potrà non seguire un tracing sequenziale, ma sembra seguire un andamento disordinato o, addirittura, far terminare il programma in modo anomalo.
Questo comportamento è dovuto al codice del programma che si sta analizzando. Infatti esistono molti modi per impedire che un programma venga tracciato.
Vediamo allora per prima cosa come funziona un debugger. Ed è qui che entrano in gioco gli INT 3 e INT 1. Questi sono i "breakpoint" e i "single stepping" interrupts rispettivamente.
Cosa succede quando settiamo un breakpoint? Il debugger legge all'indirizzo che specificate, un byte e lo mette da qualche parte in memoria. Questo byte fa parte dell'intera istruzione che si trova a quell'indirizzo. Per esempio: se l'istruzione a quell'indirizzo è "INT 13", che corrisponde in linguaggio macchina a CD13h. Il debug leggerà il primo byte, CDh, e lo salva in memoria. Il CDh sarà sostituito con un INT 3 (CCh). Così il codice apparirà CC13h. In questo modo il disassemblato sarà un INT 3 seguito da una parte di istruzione. Quando la CPU arriva a questo indirizzo, incontra un INT 3 e restituisce il controllo al debugger. Il debugger sostituirà CCh con il CDh salvato in precedenza.
Con il "single stepping" accade lo stesso. Il debug inserisce un INT 3 dopo l'istruzione che sta per eseguire.
Il problema dunque sorge quando in programma usa gli INT 3. Infatti gli int 3 possono essere riprogrammati per richiamare gli INT 21 o qualsiasi altro INT. E quindi nasce un conflitto tra il programma e il debugger che vuole usare gli

INT 3 per i suoi breakpoint. Ed un "single step" può causare un errore di protezione generale o una terminazione anomala del programma.
Per evitare che la riprogrammazione dia i sui frutti, esistono numerosi patch per Soft-Ice che impediscono ai programmi di interferire con esso.

 Appendice -A- USARE WDASM (DLLs)

Abbiamo già imparato a disassemblare i programmi nel vol. 1. Ora ci occuperemo delle Librerie Dinamiche (DLL). Infatti molto spesso la tecnica jungle ci costringe ad addentrarci in numerose dll. Una dll può essere disassemblata esattamente come un file eseguibile. L'unica differenza è che non la si può debuggare singolarmente, ha bisogno di essere appesa al file che la richiama.
Quindi per prima cosa disassembliamo il programma che la richiama.
Lo carichiamo in memoria ^Debug^ Load process (ctrl+L). E dalla finestra dell' EIP (quella di sinistra) cerchiamo la nostra dll nella lista "Active DLLs".
Doppio-click sulla dll. Et voilà... la dll è disassemblata e pronta per essere debuggata. Se la dll richiama altre dll, ripetere la procedura.
Nel prossimo volume vedremo cosa sono e come interpretare i riquadri "Regs" e "Source For Data".

 7.0 CONCLUSIONI

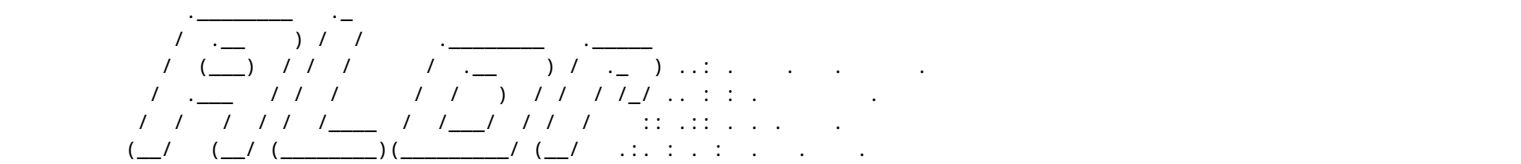
Eccoci giunti alla fine del secondo volume della "The Italian Cracking Encyclopedia". E' già in cantiere un terzo volume (probabilmente su SoftIce), quindi... Stay tuned for update !!
In fin dei conti il cracking è un'arte in continua evoluzione...
Per avere eventuali future release di questo manuale scrivete a: nz2@usa.net
oppure visitate il sito http://Alor.home.ml.org

"If you give a man a crack he'll be hungry again
 tomorrow, but if you teach him how to crack, he'll
 never be hungry again"
+ORC

Greetings to: LordKasKo (my cracking teacher...)

The other NEURO ZONE 2 members (10t8or, DK2DEnd, LordKasKo, MaPHas, Ob1, TurboSnail, XXXX)

All the Cracker on the NET from whom I have learned something.



--==> ALoR <==---------- - -   -
ICQ: 10666678   In IRC: WhiteFly

e-mail: ALoR@thepentagon.com   www: http://ALoR.home.ml.org/