The Italian CRaCKiNG Encyclopedia vol. 4

Making KeyGenerator

Scritto da

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

Dicembre 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!

Ormai lo saprete a memoria.....ma mi è necessario per pararmi il culo !!!

2.0 I TOOLS NECESSARI

Per produrre un Generatore di Serial # sono necessari:

* W32Dasm 8.9 or higher
* Soft-Ice 3.2 or higher
* Un compilatore ad alto livello (C, C++, ecc.)

3.0 PREREQUISITI

E' necessario che conosciate almeno le basi dell'assembler. Intendo almeno le istruzioni add, mul, div, mov, push, pop, call, jmp... ecc.
Se volete distribuire il vostro keygenerator dovrete creare un programma che sia in grado di elaborare l'username e che produca un codice adeguato.
Per fare questo dovreste conoscere almeno un linguaggio ad alto livello (C, C++, Pascal, ecc..).

```
------------
 4.0  INTRO
------------

Sempre piu' spesso i programmatori di shareware o trialware si affidano alla
rete per diffondere i propri programmi. Per potersi registrare e' necessario
(almeno loro credono :) pagare una somma di denaro alla casa produttrice che
ti fornira' il codice magico univoco e intestato solo a te.
Nasce quindi il key generator... un programmino che ti permette di avere il
tuo nome nel campo User: della licenza del tuo programmino preferito, e senza
aver pagato nemmeno una lira :)
Il bello del keygen e' proprio qui. Quando e' finito puoi sfornare quanti
codici ti pare !! (es. User: Micro$oft Sucks  Key: xxxxxxxxxxx :)
Un altro vantaggio del keygen sta nel fatto che il codice del prg. non e'
alterato. Questa e' un ottima cosa, specialmente negli ultimi tempi i prg. si
auto controllano (con CRC e menate varie) per vedere se sono stati modificati.
In conclusione il keygen e' il modo migliore per crackare un prg. semplice da
usare (non tanto da fare... argh..), pulito e poi fa un sacco figo :)


-------------------------
 5.0  ANALISI DEL CODICE
-------------------------

Nel vol. 3 abbiamo scoperto come trovare il codice giusto in una zona di
memoria. Ma non potete certo ripetere la procedura ogni volta che vi serve
un User nuovo....
Impareremo ora a crearci il keymaker per Winzip 6.3 (andra' bene anche per
tutte le ver precedenti e anche per la 7.0).
Disassembliamo l'eseguibile (ho scelto la ver 6.3 in modo che una parte dell
analisi e' gia' stata discussa nel vol. 3) e cominciamo.

Vi ricordate... eravamo arrivati a scoprire che:



* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004097C6(C)
|
:004097DE  lea eax, dword ptr [ebp+FFFFFDF8]
:004097E4  push eax
:004097E5  push 00471258                        <-- indirizzo del nome
:004097EA  call 004098C3                        <-- calcolo codice
:004097EF  pop ecx
:004097F0  pop ecx                              <-- provate "d ECX" :-)
:004097F1  push 0046F578                        <-- vi ricordate cos'era ?...
:004097F6  lea eax, dword ptr [ebp+FFFFFDF8]    <-- in questo istante EBP
                                                 contiene 0012F950. +FFFFFDF8
                                                 in complemento a 16 è -208.
                                                 Quindi stà mettendo in EAX
                                                 qualcosa che si trova all'
                                                 indirizzo 0012F748. Cosa ci
                                                 sarà mai a quell'indirizzo.:)
:004097FC  push eax
:004097FD  call 004465D0                        <-- parametri passati:
                                                     0046F578  il nostro codice
                                                     EAX       il codice giusto
:00409802  pop ecx
:00409803  pop ecx


Ora la parte piu' succosa non e' piu' "d ECX" oppure "d 0012F748" dove si
trova il codice, bensi' la call 004098C3 che calcola il codice esatto.
Dovremo quindi analizzare la routine all'indirizzo 004098C3 e tradurla in un
linguaggio ad alto livello per creare il nostro keygen.

Mettetevi comodi, carta, penna, martini, moooolta pazienza... Let's go...

Allora... ecco la procedura con un po di analisi a margine...


* Referenced by a CALL at Address:
|:004097EA
|
:004098C3    push ebp
:004098C4    mov ebp, esp
:004098C6    sub esp, 00000010
:004098C9    and word ptr [ebp-0C], 0000          ebp-0C = 0          2^Half
:004098CE    and word ptr [ebp-08], 0000          ebp-08 = 0          1^Half
```

```
:004098D3    mov eax, dword ptr [ebp+08]            eax = username
:004098D6    mov dword ptr [ebp-04], eax          ebp-04 = username
:004098D9    and word ptr [ebp-10], 0000          ebp-10 = 0
:004098DE    jmp 004098F3


Fondamentalmente inizializzazione delle variabili. ebp-0c rappresenta la zona
di memoria dove sara' allocata la seconda parte del codice, ebp-08 sara' per
la prima. Infatti il codice e' formato da due numeri esadecimali da 4
caratteri ciascuno. Ci aiuta a scoprirlo la stringa "%04X%04X" (per chi si
intende di C) Vedi sotto dove e' utilizzata.


* Referenced by 00409915(U)
|
:004098E0    mov ax, word ptr [ebp-10]        \
:004098E4    add ax, 0001                      | count = count + 1
:004098E8    mov word ptr [ebp-10], ax        /
:004098EC    mov eax, dword ptr [ebp-04]      \  username[count]
:004098EF    inc eax                           |
:004098F0    mov dword ptr [ebp-04], eax      /  ALoR --> LoR --> oR --> R

* Referenced by 004098DE(U)
|
:004098F3    mov eax, dword ptr [ebp-04]      \
:004098F6    movzx eax, byte ptr [eax]         | if username[count] == 0
:004098F9    test eax, eax                     |                jmp fine_loop
:004098FB    je 00409917                      /
:004098FD    mov eax, dword ptr [ebp-04]         eax = username
:00409900    movzx eax, byte ptr [eax]           eax = username[count]
:00409903    movzx ecx, word ptr [ebp-10]        ecx = count
:00409907    imul eax, ecx                       eax = username[count] * count
:0040990A    mov cx, word ptr [ebp-0C]        \
:0040990E    add cx, ax                        | 2^Half = 2^Half + ax
:00409911    mov word ptr [ebp-0C], cx        /
:00409915    jmp 004098E0                     LOOP


Queste linee di codice costituiscono il loop per la creazione della seconda
parte del codice.
ebp-10 sara' in nostro counter. ebp-04 l'username. ebp-0c la parte di codice
ATTENZIONE: quando si calcola 2^Half = 2^Half + ax bisogna tenere presente che
qui si opera su registri a 16 bit (ax, cx) e non a 32 bit (eax, ecx).



* Referenced by 004098FB(C)
|
:00409917    mov dword ptr [00471C5C], 00000001   <-- Ma serve ?? imho NO :)
:00409921    mov eax, dword ptr [ebp+08]          eax = username
:00409924    mov dword ptr [ebp-04], eax       ebp-04 = username
:00409927    jmp 00409930

* Referenced by 00409956(U)
|
:00409929    mov eax, dword ptr [ebp-04]      \  username[count]
:0040992C    inc eax                           |
:0040992D    mov dword ptr [ebp-04], eax      /  ALoR --> LoR --> oR --> R

* Referenced by 00409927(U)
|
:00409930    mov eax, dword ptr [ebp-04]      \
:00409933    movzx eax, byte ptr [eax]         | if username[count] == 0
:00409936    test eax, eax                     |                 jmp fine_loop
:00409938    je 00409958                      /
:0040993A    push 00001021                             stack 0x1021
:0040993F    mov eax, dword ptr [ebp-04]         eax = username
:00409942    movzx ax, byte ptr [eax]            eax = username[count]
:00409946    push eax                                  stack username[count]
:00409947    push [ebp-08]                             stack 1^Half
:0040994A    call 00409980                             CALL calcola --->
:0040994F    add esp, 0000000C                   esp = esp + C
:00409952    mov word ptr [ebp-08], ax           1^Half = ax
:00409956    jmp 00409929                     LOOP


Loop per il calcolo della prima parte di codice (la piu' complicata).
ATTENZIONE: il vero calcolo e' effettuato dalla CALL 00409980. e i suoi
parametri gli sono passati attraverso le PUSH.
```

Quindi i suoi parametri saranno: 0x1021, username[count] e 1^Half
Il ritorno e' messo in ax

```
* Referenced by 00409938(C)
|
:00409958    mov ax, word ptr [ebp-08]        \
:0040995C    add ax, 0063                      | 1^Half = 1^Half + 0x63
:00409960    mov word ptr [ebp-08], ax        /
:00409964    movzx eax, word ptr [ebp-0C]        eax = 2^Half
:00409968    push eax                                        stack 2^Half
:00409969    movzx eax, word ptr [ebp-08]        eax = 1^Half
:0040996D    push eax                                        stack 1^Half

* Possible StringData Ref from Data Obj ->"%04X%04X"
                                  |
:0040996E    push 00463954                                   stack "%04X%04X"
:00409973    push [ebp+0C]                                   stack ebp+0C
:00409976    call 004452E0                                   CALL 004452E0
:0040997B    add esp, 00000010                  esp = esp + 10
:0040997E    leave
:0040997F    ret                                END
```

Parte finale. Aggiunge 0x63 alla prima parte (okkio ai 16 bit) e unisce le due
parti producendo un numero della forma: %04X%04X es: 11aa0220.

Analizziamo ora il calcolo della prima parte di codice :

```
* Referenced by a CALL at Addresses:
|:0040994A   , :00409ACB
|
:00409980   push ebp
:00409981   mov ebp, esp
:00409983   push ecx
:00409984   mov ax, word ptr [ebp+0C]        \
:00409988   shl ax, 08                        | lettera = username[count] << 8
:0040998C   mov word ptr [ebp+0C], ax        /
:00409990   and dword ptr [ebp-04], 00000000    i = 0
:00409994   jmp 0040999D
```

Prende i parametri dallo stack e inizializza il counter (ebp-04).
Il parametro username[count] (qui ebp+0C) e' shiftato a sinistra di 8, cioe'
moltiplicato per 0x100 (256).

```
* Referenced by 004099DE(U)
|
:00409996   mov eax, dword ptr [ebp-04]         \
:00409999   inc eax                              | i = i + 1
:0040999A   mov dword ptr [ebp-04], eax         /

* Referenced by 00409994(U)
|
:0040999D   cmp dword ptr [ebp-04], 00000008    | if (i < 8) jmp fine_call
:004099A1   jge 004099E0                        |
:004099A3   movzx eax, word ptr [ebp+08]          eax = 1^Half
:004099A7   movzx ecx, word ptr [ebp+0C]          ecx = lettera
:004099AB   xor eax, ecx                          eax = temp ^ lettera
:004099AD   and eax, 00008000                     eax = eax & 8000
:004099B2   test eax, eax                       | if (eax == 0) jmp 004099C8
:004099B4   je 004099C8                         | else {
:004099B6   movzx eax, word ptr [ebp+08]            eax = 1^Half
:004099BA   shl eax, 1                              eax = eax << 1
:004099BC   movzx ecx, word ptr [ebp+10]            ecx = 0x1021
:004099C0   xor eax, ecx                            eax = eax ^ 0x1021
:004099C2   mov word ptr [ebp+08], ax               1^Half = ax
:004099C6   jmp 004099D3                        | }
```

In questo istante ebp+10 e' il parametro passato 0x1021. Attenzione ai soliti
16 bit. La costruzione dell'if non e' palese... ma ci si arriva.

```
* Referenced by 004099B4(C)
|
:004099C8   mov ax, word ptr [ebp+08]           \
:004099CC   shl ax, 1                            | 1^Half = 1^Half << 1
:004099CF   mov word ptr [ebp+08], ax           /

* Referenced by 004099C6(U)
|
:004099D3   mov ax, word ptr [ebp+0C]           \
:004099D7   shl ax, 1                            | lettera = lettera << 1
:004099DA   mov word ptr [ebp+0C], ax           /
:004099DE   jmp 00409996                     LOOP

* Referenced by 004099A1(C)
|
:004099E0   mov ax, word ptr [ebp+08]             ax = ebp+08
:004099E4   leave
:004099E5   ret                              END
```

Fine della procedura e ritorno del codice in ax (16 bit).

 6.0 SCRITTURA DI UN KEYMAKER

Dopo aver analizzato a fondo la procedura in linguaggio assembly l'80% del lavoro e' ormai compiuto. Ora bastera' tradurre, con qualche accorgimento, il codice assembly in uno ad alto livello quale il C o il C++ e il gioco e' fatto

Ancora qualche sforzo... siamo in dirittura d'arrivo...

un altro martini e via...

Tralascio qui la scrittura del main e dell'input dell'username. Trattero' solo le due procedure di calcolo delle rispettive parti di codice.

*** Procedura calcolo seconda parte ***

```
long km2(char *User)
{
long code2 = 0;                                          /* ebp-0C */
int count = 0;                                           /* ebp-10 */
long lettera;

for (count = 0; count < strlen(User); count++) {         /* LOOP */
   lettera = User[count];
   lettera *= count;
   code2 += lettera;
}

return code2;
}
```

*** Procedura calcolo prima parte ***

```
long km1(char *User)
{
long code1 = 0;                                          /* ebp-10 */
long temp = 0;
int i, count = 0;                                        /* ebp-04 */
int lettera;


for (count = 0; count < strlen(User); count++) {
   lettera = User[count];
   lettera <<= 8;

   for(i = 0; i < 8; i++) {
      temp = code2;
      temp ^= lettera;
      temp &= 0x8000;
```

```
      if (temp == 0) {
         code2 <<= 17;                              /* shift a 16 bit */
         code2 >>= 16;
      } else {
         code2 <<= 17;                              /* shift a 16 bit */
         code2 >>= 16;
         code2 ^= 0x1021;
      }
      lettera <<= 17;                               /* shift a 16 bit */
      lettera >>= 16;
   }
}

code2 += 0x63;

return code2;
}
```

ATTENZIONE: per effettuare lo shift di 1 a sinistra a 16 bit su una variabile
a 32 bit. Ho shiftato di 16 + 1 a sinistra e successivamente di 16 a destra.

esempio:

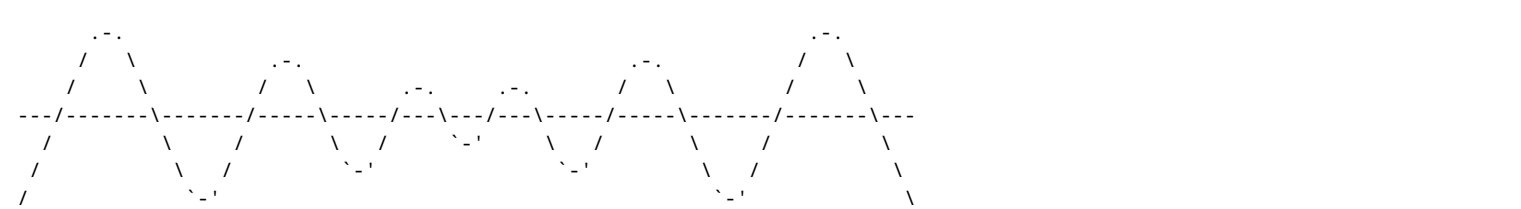
```
      var16:      1234   ==        var32:  0000 1234
 var16 << 1:      2340   !=   var32 << 1:  0001 2340

                             var32 << 17: 2340 0000
 var16 << 1:      2340   ==   var32 >> 16: 0000 2340
```

Ecco fatto !! We reached the goal !!!
Dopo una dura battaglia e la bottiglia del martini ormai vuota, la testa che
gira sotto effetto dell'alcool... Il nostro keymaker e' pronto.

7.0 CONCLUSIONI

Per il momento non ho idee su come continuare l'enciclopedia. Quindi se avete
suggerimenti per il vol 5 contattatemi pure via mail ALoR@thepentagon.com
Per avere eventuali future release di questo manuale scrivete a: nz2@usa.net

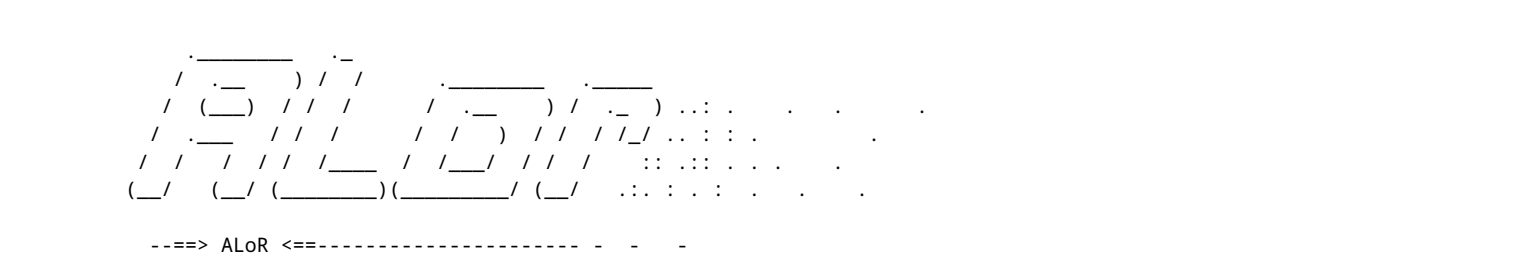
"If you give a man a crack he'll be hungry again
tomorrow, but if you teach him how to crack, he'll
never be hungry again"
+ORC



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

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

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



ICQ: 10666678 In IRC: WhiteFly

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

==========