HOW TO BECOME A SERIAL KILLER

(A Hacker's Guide to Reverse Engineering Serial Number Algorithms)

by =-BOOK-WORM->

If you've ever cracked an application before, you can understand the thrill of the quest, but if you've gone as far as figuring out how serial numbers can be generated, it puts you more into the mind frame of a secret agent, code breaking for the FBI. In some cases it truly presents some of the toughest puzzles you'll ever encounter (except for perhaps, the "Rubik's Cube"). There is only one quality that is a must-have here....DETERMINATION! I can not stress that enough. If you're not willing to put some time into it, stop reading right now! So the magic word for today folks is??? - DETERMINATION!!!

My intentions for writing this article are to provide steps and examples for those who already possess the following skills:

- Knowledge in using a debugger (My favorites in order: TMON, Jasik's, MacsBug)
- Macintosh 68k assembler (As long as you have a manual, you should be OK)
- Bitwise operations (OR, AND, XOR, etc...)
- Basic Algebra (Ha! And you thought you'd never use it!)
- Determination (The magic word)

(Skills in operating a printer, doodling on paper, etc... are already assumed.)

I will be using "sn" to refer to serial number, registration code, etc... throughout this text.

THE FOUR QUESTIONS

Only four questions need to be answered in solving each case:

1. Where is it?
2. What does it do?
3. How can it be reversed?
4. Do the generated numbers work?

Knowing this, we will explore each question in detail.

1. WHERE IS IT?

Finding the routine can often be a challenge. Luckily, the _GetIText trap will solve this question for you more often than not. In most cases, each field in the dialog box will have a separate _GetIText issued to retrieve its contents. The following technique can be used in these instances:

Technique 1
----
* Get to the application's registration screen
* Enter debugger
* Set a trap for _GetIText
* Exit the debugger
* Type something in all non-sn fields
* Type "1234567890" into the sn field
* Press "OK" (or equivalent)
(At this point you should enter the debugger. If not, read further for other techniques.)
* View the contents of the area pointed to by register A1
* If it contains "1234567890", start single stepping
* If it does not, continue execution until it does, then start single stepping

When starting to learn, you should single step the entire way from this point. As you become more experienced, you'll learn time saving skills by identifying specific library routines like "pascal to C string" functions which you may simply jump over.

In more challenging situations where _GetIText is not used, you may need to trap _TEKey. If pressing the return key is permitted in lieu of the mouse click to accept the input, use:

Technique 2
----
* Get to the application's registration screen
* Type something in all non-sn fields
* Type "1234567890" into the sn field
* Enter debugger
* Set a trap for _TEKey
* Exit the debugger
* Press return
(At this point you should enter the debugger)
* Start single stepping

If a mouse click is needed to accept the input, use:

Technique 3
----
* Get to the application's registration screen
* Type something in all non-sn fields
* Type "123456789" into the sn field
* Enter debugger
* Set a trap for _TEKey
* Exit the debugger
* Type "0"
(At this point you should enter the debugger)
* Start single stepping

Each case is different. In some cases, the developer may be generating a checksum "as you type" and therefore Techniques 2 or 3 are necessary. In most others, he checks after you press return where Techniques 1 and 2 would suffice. Following code after a _TEKey can be tedious. I often set breakpoints to _BlockMove after I return from a _TEKey break. Next, I check the area pointed to by register A0 after each _BlockMove is encountered until I find the sn which I entered. Sometimes you get lucky and can treat a _BlockMove break like a _GetIText break from there.

Other techniques involve trapping _ModalDialog and finding where it will go when the user presses OK. I rarely use that technique anymore as there may be user functions attached to _ModalDialog which process each keystroke and if not, _GetIText will put you further into the code.

"Where is it?" can be defined as "Where does it first reference anything I've typed into the dialog box?". The sure-fire way to find it is to maintain the single stepping up to that point.

I'll be using Knot 3.6 as an example as its sn routine is not too extensive but still requires some thinking in its reversal. The following code is found after using Technique 1 (and many single steps later):

```
         MOVEA.L 8(A7),A0
         PUSH    #1  ; = StringToNum
         _Pack7
         MOVEA.L 4(A7),A0
         MOVE.L  D0,(A0)
```

This is the first occurrence of reference to the sn. Here it converts the sn string into a number for further processing. Now we're ready to proceed to question number two.

2. WHAT DOES IT DO?
===================

The following are favorites to developers in checking your sn:

* Is the length correct?
* Does it contain specific characters at predefined locations?
* Does a calculation on some part of it result in the another part?
* Does a calculation on other fields result in a part of it?
* Does a lookup table on one part result in another?
* Who cares what the user types? (mission accomplished!)

This is the step where you need to get out your number two pencils and tablet of paper. As you single step through the code, you must write down all that happens. I tend to use tree diagrams stemming from the sn which I write at the top of the page with separation between each character. Sometimes I need to draw arrows showing characters which are swapped. Sometimes I write replacement characters above them. Many times I have lines streaming down each character, or set of characters, resembling long term division but involving bitwise and/or other operators. Use whatever works best for you but be sure to take down everything in a format legible at least to yourself, or you'll be sorry later.

Upon cruising through the code past the initial string-to-number routine, we find this:

```
         PEA     vjb_2(A6)
         JSR     ITSGOOD            ; jump somewhere
         POP.B   D0                 ; retrieve the result code
         BEQ.S   ljb_2              ; if result code = 0, jump to error alert
         SUBQ    #2,A7
         PEA     vjb_2(A6)
         PEA     glob66(A5)
         CLR.L   -(A7)
         JSR     GETPREFS
         POP.B   D0
         MOVE.B  D0,vjb_1(A6)
         MOVE.B  #1,glob43(A5)
         SUBQ    #2,A7
         PUSH    #$80
         PEA     0
         _Alert  ; (alertID:INTEGER; filterProc:ProcPtr):INTEGER
         POP     D0
         EXT.L   D0
         MOVE.L  D0,vjb_2(A6)
         PUSH.L  $2988(A5)
         JSR     DO_CLOSE
         BRA.S   ljb_3
ljb_2    SUBQ    #2,A7
         PUSH    #$81
         PEA     0
         _Alert  ; (alertID:INTEGER; filterProc:ProcPtr):INTEGER
```

At this point we know that "ITSGOOD" must NOT return a zero.
Now let's see what lurks inside "ITSGOOD":

```
ITSGOOD  LINK    A6,#0
         PUSH.L  A2
         MOVEA.L param1(A6),A2
         CLR.B   funRslt(A6)        ; Assume an invalid sn (prime a zero)
         TST.L   (A2)               ; Rule 1
         BEQ.S   liy_1              ;   "
         MOVE.L  (A2),D0            ; Rule 2
         ANDI.L  #$100,D0           ;   "
         TST.L   D0                 ;   "
         BNE.S   liy_1              ;   "
         CMPI.L  #$186A0,(A2)       ; Rule 3
         BLT.S   liy_1              ;   "
         MOVEQ   #31,D0             ; Rule 4
         AND.L   (A2),D0            ;   "
         ADDI.L  #$4531,D0          ;   "
         MOVE.L  D0,D1              ;   "
         MOVE.L  (A2),D0            ;   "
         JSR     proc13             ;   " ==> proc13
         CMPI.L  #$9D,D0            ; Rule 4 continued
         BNE.S   liy_1              ;   "
         MOVE.B  #1,funRslt(A6)     ; All checks are valid!
liy_1    POP.L   A2
         UNLK    A6
         POP.L   (A7)
         RTS

proc13   TST.L   D1                 ; Rule 4 continued
         BGE.S   lao_1              ;   "
         NEG.L   D1
lao_1    TST.L   D0                 ;   "
         BLT.S   lao_2              ;   "
         JMP     proc12             ;   " ==> proc12
lao_2    NEG.L   D0
         JSR     proc12
         NEG.L   D0
         RTS

proc12   MOVEM.L D2-D3,-(A7)        ; Rule 4 continued
         MOVEQ   #2,D2              ;   "
         JMP     data11-2(D2.W*2)   ;   " ==> data11

data11   BRA.S   lan_1              ; Rule 4 continued
         DIVUL.L D1,D1:D0           ;   "
         MOVE.L  D1,D0              ;   "
         MOVEM.L (A7)+,D2-D3        ;   "
         RTS                        ;   " ==> return back to ITSGOOD
```

In viewing the code, we can now construct a set of rules which must be met for proper sn validation:

```
Rule 1 : sn must NOT be a zero
Rule 2 : (sn & $100) = 0
Rule 3 : sn >= $186A0
Rule 4 : sn MOD ((sn & 31) + $4531) = $9D
```

Now we are ready to answer question three.

3. HOW CAN IT BE REVERSED?
==========================

This question rarely has the same answer. I've seen quite a few different techniques used in sn checksumming. The main thing to remember here is, good notes = success. I find it useful at this point to take a deep breath, view my notes, and ask myself "why?". Why is it flipping every 5th bit or why does it repeat a certain pattern over and over again? In answering these questions, you may find a simple way to repeat that capability in a more simplistic form. Another thing to keep in mind is that there are often multiple ways to derive the same sn. I will demonstrate this by providing two separate routines for reversing the Knot sn checksum.

In viewing the rules which were derived from question two, we find that rule 3 will allow us to disregard rule 1. Rule 4 is the toughest one and will require us to dust off our old algebra books for solutions to simplification or take the easy way out. First let's try the easy way, or better described as the "brute force" method:

```
FOR sn = $186A0 TO mymaxno
  IF ((sn & $100)=0) AND (sn MOD ((sn & 31) + $4531)=$9D) THEN PRINT sn
NEXT
```

Rule 1 : (Disregarded in lieu of rule 3)
Rule 2 : Applied in the first half of the "IF" statement
Rule 3 : Applied by the outer loop (sequentially increment sn from $186A0)
Rule 4 : Applied in the second half of the "IF" statement

Although this method will work, it's slow. Trying every number is never a good solution as the distance between valid numbers may be great. So let's look a little deeper to find a better solution.

Let's see now:

Rule 4 : sn MOD ((sn & 31) + $4531) = $9D

...is the same as saying:

Rule 4 : sn = x * ((sn & 31) + $4531) + $9D

Also, by sheer knowledge, we know that (sn & 31) can have only 32 possibilities (0-31). So let's reinstate the formula where "p" may be any one of the 32 possibilities:

Rule 4 : sn = x * (p + $4531) + $9D

Now we can simply create an algorithm with nested loops. The outer loop will control "x" and the inner loop will control each iteration of "p" (0-31):

```
FOR x=1 TO mymaxno
  FOR p=0 TO 31
    sn = x * (p + $4531) + $9D
    IF sn >= $186A0 AND ((sn & $100) = 0) AND ((sn & 31) = p) THEN PRINT sn
  NEXT
NEXT
```

Notice the last part of the "IF" statement. This is where we verify that p does, in fact, equal (sn & 31).

Using this second approach greatly reduces the time needed to generate numbers as we are skipping through all possibilities by leaps and bounds.

Now we are ready to answer the final question.

4. DO THE GENERATED NUMBERS WORK?
=================================

This is the easiest question to answer as you can simply type them in to check the results. I recommend trying the lowest and highest sn followed by a few in between and a series "in a row". If your notes and programming are good, they usually all work fine. If there is a bug in your programming, they are usually all wrong or the lower and/or upper limits are wrong.

In this case, after running both algorithms, we arrive at the same results. There is, of course, a noticeable difference in speed between them.

ENDING COMMENTS

I've taken you on a journey through a simple situation. There are, however, many more difficult routines being used in todays sn checksums. You must keep in mind that nothing is impossible through determination. No matter how you fold a piece of paper, it's always possible to unfold it once again. Some folds may be tucked in deep and difficult to pull out. There may be times you encounter hundreds of folds. But no matter how much time is put into folding that sheet of paper, someone else can always unfold it. There's no reason why you can't be that person!

Let the puzzles begin,
=-BOOK-WORM->