(Continued from previous chapter)

More On Using MacsBug

The 's' and the 'so' MacsBug Commands
Often, a registration routine will call the ModalDialog trap to find out what the user's doing. When the user hits ok, it will call a subroutine to determine whether the code is valid, and the subroutine will return a "yes or no" value. In MacsBug, you can step through instructions to see what's going on. You have two choices. You can either step through every instruction, which will be really tedious unless you are pretty close to what you're looking for, or you can step through only the instructions in the current routine, stepping over toolbox traps and subroutines. This is good for getting a general understanding of what the program is doing. To step through individual instructions, use the 's' (step) command. To step over subroutines and traps, use the 'so' (step over) command. Hitting return will repeat the last command executed, so you don't have to type 'so' over and over. You can also hit escape to see the Smac screen; hit escape again to get back to MacsBug. After a JSR has been executed, the subroutine will return to the original routine with an rts instruction (return from subroutine).

When using the 's' command, MacsBug will execute the current instruction and allow the user to interact with MacsBug immediately afterwards. If the current instruction is a JSR or BSR, 's' will execute the JSR or BSR instruction and then show you the first instruction in the subroutine called by the JSR or BSR. This also applies to toolbox traps. The 's' command will show the MacsBug user every single instruction the computer ever executes ('kay, if you're a guru reading this, you don't get to see interrupts, but who cares?).

When using the 'so' command, MacsBug will execute the current instruction *and* everything associated with it, then return control to the user afterwards. If the current instruction is a JSR or a BSR, 'so' will execute the JSR or BSR, execute the subroutine called by the JSR or BSR, execute the RTS at the end of the subroutine, then return control to the MacsBug user with the current instruction set to the one that followed the JSR or BSR. Otherwise, 's' and 'so' are equivalent.

The 'br' and 'brc' and 'gt' commands
Lets say you want your program to run until it gets to a certain

place and then drop into MacsBug. You can set a breakpoint for some address in memory. When the program counter (PC) is equal to the address of one of your breakpoints, you will drop into MacsBug. This is useful if you've eliminated some section of your program as being irrelevant to your crack and you don't want to have to step through it. To set a breakpoint, the syntax is br <the address at which to break>. Keep in mind that you can use expressions here, like "br pc+4", which will break at four bytes beyond the current instruction.

When using the 'br' (BReak point) command, execution will *always* stop when the pc is equal to your breakpoint. If you want clear a breakpoint, you can use the 'brc' (BReak point Clear) command. This can be brc <the address> to clear a particular breakpoint or just brc to clear 'em all.

If you want to break at some location only one time, you can use the 'gt' (Go Till) command. This is exactly equivalent to setting a breakpoint, running till you get to it, then clearing it.

The 'g' Command
Typing 'g' will continue execution normally until a breakpoint is encountered.

Displaying and Setting Memory
You can look at or set the contents of memory. To look at 16 bytes of memory, use dm <the address> (dm stands for display memory). To look at only a byte, word, or long, use db, dw, or dl, respectively. You can set a byte, word, or long by using sb <the address> <the byte>, sw or sl, respectively (sb stands for set byte). This can be used to see whether the registration code you typed is inside of an address being manipulated by the program. It can also be used to change stuff on the fly.

Other MacsBug Commands
Finally, you can try to do an emergency exit from the program with es (Exit to Shell), you can restart the computer with rs (ReStart), or reboot (with the memory check and all the stuff that makes it take 14 years) with rb (ReBoot). You'll probably crash the computer quite a few times trying to krack programs, so these commands are good ones to know. In fact, even if you don't use MacsBug for anything else, it's worth having just for these commands. The 'es' command, for example, is more robust than

doing a force quit from a program with cmd-opt esc, and using rs is quicker than manually restarting the computer. These commands are not strictly relevent to kracking programs, but they're pretty good to know.

Number Conversion
MacsBug will translate hex to decimal for you, just type in a hexadecimal number and you'll get the decimal prefixed with a #. For example, if I type 524C (a hex number), I get

524c = $0000524C
#21068
#21068
'∑∑RL'
(between 20k and 21k)

This tells me that the expression I typed in (524C) is equal to 524C hex, 21068 unsigned decimal, 21068 signed decimal, '∑∑RL' ascii and is between 20 and 21K in memory size. You can also type simple equations and get the same type of output.

If you want to convert a decimal number to hex, you can type the decimal number preceeded by a '#'. For example, typing '#10' will tell me that 10 decimal is equal to 0000000A hex.

Doing The Krack

Allrighty, enough preamble crap. Here's the basic strategy revisited. You will fill in the text fields in the registration window with whatever you want, set an a-trap break for ModalDialog, and step through the code till you find where it says "yes or no" to the good registration question. Here's how I would do this, you can do it however ya want.

Type everything you want in the text fields except the very last character you intend to type.

Drop into MacsBug (I use cmd-power key to do this) and type "atb ModalDialog" to set an a-trap break on the ModalDialog trap. The next time ModalDialog is encountered, you will drop into MacsBug. You don't type all the characters because when you originally

drop into MacsBug, you will almost certainly already be inside the ModalDialog trap, and you want to be outside of it.

Type 'g' to continue execution normally and type the last character into the program's text field. At this point, you should drop into MacsBug, and the next instruction should be ModalDialog. If it's not or you don't drop into MacsBug, you've got to try a different toolbox trap, maybe DialogSelect.

Type 'so' to step over the modal dialog trap. This will let you do one thing (like click the OK button or hit return) and then will drop you back into MacsBug at the instruction following ModalDialog.

Click the ok button, and you're back in MacsBug. You'll use 'so' to step over instructions looking for that "yes or no" check. You may try using 'dm' to display the memory that the instructions are dealing with. For example, if an instruction uses -$0016(A0), you could try 'dm a0-16' to see the memory. If the first eight bytes of the memory displayed by 'dm' look like an address, you could try doing a 'dm' on the address in case it uses double indirection. Somewhere along the line, you should see whatever you typed in as your serial number. This'll mean you're on the right track. You can also look for either the GetDialogItem or GetDialogItemText toolbox traps. These get information from a window (such as the serial number you typed). Anyway, if you persevere and think about what you're seeing, eventually you may find something that looks like either the example given in the explanation of the MOVE instruction, or like the following

TST.B
D0
Bcc.s
<somewhere>

where <somewhere> is the location that will be branched to. <Somewhere> will not be surrounded by <>, it may look like 'CODE 0001'+002A.

This is testing a yes or no. D0 is a data register, it could be D(some other number). If it branches (see the branch instruction), try not branching and then type g to continue normal execution and vice versa. If you're lucky, you'll get the lovely screen that says "Thanks for registering." If you play around for more than 200 hours and haven't found it, guess you'll have to use a different approach.

Another way to find the all-powerful "yes or no" check is to step over (so) instructions until you see the "Wrong Code, Bub" message. Make a note of the address at which this happened. Was there a conditional branch not too long before? That may be your branch. If it happens inside of a subroutine (i.e., the last instruction you stepped over was JSR or BSR), the check *may* happen inside the subroutine. However, the subroutine may just be the DisplayAnnoyingWrongCode subroutine. You can 'so' until you get to the subroutine, then 's' once to get inside it, then continue to 'so' till you get the "Wrong Code Moron!" message. Repeat as necessary, do not stir until boiling.

The "Hey, SuckBag - You're Trying To Krack Me" message will usually be executed by the Alert toolbox trap. If you're using the above method and end up at the Alert trap, you've missed the check.

Allrighty, You've Kracked It, Now What?

Have I Kracked It?
If you have found the branch instruction that allows you to get a valid registration and continued execution results in "Hey, Thanks For Registering," you have kracked the program. If MacsBug is listing offsets next to the conditional branch you found, make a note of the routine and the offset (see the explanation of the MacsBug display). If not, write down as much machine language from that point on as you can (I usually write down about 20 bytes). If the program is now kracked, you can just say to forget it and leave it at that. However, if you want to krack it for someone else, you'll have to actually change the program's code.

Finding The Place To Change
To change code, you'll use ResEdit and the CodeEditor. You'll find the branch instruction that determines "yes or no" and change it so it either always branches or never branches, depending on what kracks the code. So, into ResEdit you go, and open up the resource corresponding to where the branch instruction is. If you can't figure out how to open files in ResEdit, this tutorial may be more applicable to the guy in the

next cell over from you. For example, if MacsBug told you that the branch instruction looked like this:

'CODE 000A 29DE TCL Critical'

; Will Branch
+02B36
05E4B886
*BEQ.S
'CODE 000A 29ätical'+02B54
; 05E4B8A4

|671C

you will look in the CODE resource with ID 10 (000A hex - see More MacsBug Stuff).

So, open CODE resource 10. If you have the CodeEditor, you'll see an assembly language version of the resource. From the "Resource" menu, select "Open Using Hex Editor". This will give you the raw hex and ascii version you would get if you didn't have the CodeEditor. From the "Find" menu, choose "Offset", and type in the offset to the branch statement that you wrote down earlier. In the example above, the offset would be 2B36.

If MacsBug was *not* listing offsets, you would choose "Find Hex" from the "Find" menu and type in the machine language you had written down, exactly and with no spaces. You may have to try several different resources before you find a match. Even when you *do* find a match, it's remotely possible that this is still the wrong resource, although it's highly unlikely.

Changing The Code
Okay, here's where you actually change the code. In machine language, all branch statements begin with a 6. To make the statement always branch, change 6x, where x is some number, to
60. If you go back into the Code Editor window, you should see that the statement has changed to bra.s. Yer done.

If you want to make the statement never branch, you need to get rid of the instruction. You can't just delete it, because the

software you are cracking uses offsets to determine where to branch to for other things, and you will almost certainly make those offsets invalid. So, you have to replace the instruction with something else. When you look at the branch instruction in the code editor, you will see the machine language version on the far right side of the window. You need to determine how many words the instruction uses. It will be either one or two. You will replace all the words of the instruction with 4e71. 4e71 is the machine language version of the NOP assembly instruction (No OPeration). Make sure to replace both words if the instruction uses two, otherwise you'll crash the computer when you try to run the software. And again, you're done.

More Cool Stuff To Do With The CodeEditor

The Code Editor is pretty seriously powerful. Here are some of its features. The Code Editor window and the hex editor window stay in sync. In other words, if you select something in the code editor window, then switch to the hex editor, it will be selected there too. If you change a selection in the hex editor window, then click on the code editor window, your changes will be reflected immediately. The other biggie is that you can find all of the references to an instruction. Lets say that the program has a subroutine that checks the preferences file to verify that the serial number stored in it is correct, and the program calls the subroutine a whole bunch of times throughout execution. This is a typical "make it tougher to crack the program" strategy. Well, click once on the address of the start of the subroutine in the code editor window, and the Goodies menu will give you a listing of all the places in the code where that subroutine is called. So, you can just change the code to skip all the checks, and you don't have to do anything with MacsBug except find the subroutine in the first place (my editor says "Yippee" to this).

Additional Reading

Wellp, I guess that's about it. For more info on assembly, pick up any of the three trillion books about it. My reference is "Programming the 68000" by Steve Williams. There is also a handy index card made by Motorola that has everything pertinant in the book concatenated onto it. Wish I could find mine. For more info on toolbox traps, check the Inside Smacintosh books on Apple's web site, or pick up one of the three and a half trillion "Programming the Smac" books kicking around. For more info on MacsBugs, try "Debugging Macintosh Software With MacsBug" by Othmer Straus.

Ego Bolstering For You

And, for the meek among you, no one taught me how to do any of this, I just figured it all out, so no excuses please. All it takes is patience and the intelligence of a lemur. If you think a lemur is *really* smart, maybe you should try gardening or basket weaving, or maybe even froggie taxidermy.

Now I'm Lame, So An Extra MacsBug Example

Okay, here's an extra added bonus. I played the way kool move of crashing my computer about half way through writing this (version 1.0, that is), and I hadn't saved any of it. Whooooops. Incidentally, I crashed it trying to get a particular MacsBug listing, which just goes to show how dangerous MacsBug can be, even if you *do* understand it. Anyway, I wasn't too enthusiastic about retyping the whole thing, so I fished it out of ram with MacsBug, and I'm gonna tell ya how I did it.

Obviously, as I'm typing this, it must be stored somewhere. It's stored in RAM, and to get it back, you just have to find where in RAM it is, and pop it onto disk. So, in MacsBug, use the 'hz' command (Heap Zones) to display all the heaps. A heap is a portion of memory that the Smacintosh allocates to individual programs. Now, I'm using BBEdit to type this, and the 'hz' command shows me the BBEDit heap, among others. For some weird

reason, however, I didn't find it in the BBEdit heap, so we'll just search all of application memory. Here is a typical 'hz' display from my computer, even as I type this!

Heap zones
32 4916K 00002000 to 004CF32F SysZone^ TheZone^
32 3K 000021D0 to 00002E53 !
32 9K 000C2E30 to 000C5623
32 190K 0026E950 to 0029E343
32 96175K 004CF330 to 062BB023
32 7K 05917880 to 059197F3
32 60K 0591A050 to 059290F3
32 2K 05929890 to 0592A3B3
32 9K 0592A3C0 to 0592C9C3
32 5K 0592D1C0 to 0592E813
32 2906K 059B9250 to 05C8FCD3 ≥NewsWatcher≤
32 999K 05D2F600 to 05E294A3 ≥ResEdit≤
32 2931K 05E33CB0 to 06110983 ≥AcrobatÅ Reader 2.1≤
32 255K 05ECE230 to 05F0E223
32 577K 06080290 to 06110963

32 914K 0613F200 to 06223DE3 ≥BBEdit 4.0≤ ApplZone^
TargetZone
32 142K 0623D490 to 06260F63
32 148K 06261530 to 06286643 ≥Finder≤
32 29K 0627DF90 to 06285783
32 12K 062990B0 to 0629C143 ≥Queue Watcher≤
32 20K 0629F030 to 062A4423 ≥FaxMonitor≤

You can see all the programs I'm running right now. The heaps that are indented reside inside the one above them that's not indented. So, all my applications reside inside the big 96175k heap. This is the Process Manager heap; it's used to manage applications. So, I'm gonna search the Process Manager heap. I need to know the starting address (4cf330), and the size of the heap. I can find the size by typing 62bb023-4cf330 (end - start). I get 5debcf3 back as the size of the heap. Now, I use MacsBug's Find command ('f'). You can type "? f" to see how it's used. Basically, it's f <start address> <number of bytes to search> '<the text to find>'. So, to find my document, I type

f 4cf330 5debcf3 'So, to find my document'

and after waiting what seems an extraordinary amount of time, MacsBug tells me where in memory that string is. Now, I just need to find the beginning and the end. So, I use the 'dm' (Display Memory) command to view memory before and after that address, till I've found the beginning and the end. I'll call

the beginning address x and the end address y. Now, I need to know the size of my document, so I type y-x and I get the size, which I'll call z. Time to save the whole thing to disk. Type "log <filename>" to begin logging all MacsBug output to disk (instead of <filename> you just type the name - duh). Now, I type "dma x z", where x is the beginning address I found and z is the length I found, to display memory as text from my starting address through the end of my document. Finally, I type "log" to close the log file. And, that's it. When I restart, I'll have on my desktop a file called <filename> that contains my document in pure text format.

Of course, if I'd been doing this in Microsoft Word, I'd have a ton of weird formatting characters stuck in the middle of everything, but fortunately, I'm not.

Anyway, hope this helps people, hope everyone saw the way kool lunar eclipse last night, and happy kracking!

Smeger September 27, 1996