Yes, that's right! It's... The Kool Krack Tootoriul

by smeger

version 1.1 November 9, 1996

but wait, now how much would you pay?

operators are standing by

(best results will come from viewing this file in BBEdit with Monaco 9pt)

Who's It For?

This tutorial is for people who have no idea how to crack programs and have no idea how to program anything. It contains a description of the assembly language commands usually used to crack, a description of the software tools used to crack, and an illustration of the technique of cracking. The illustrative technique gives an example of cracking a program that displays an annoying "Register Me" message and requests a registration code. However, the information given should be extensible to any cracking situation. This tutorial should give a novice enough information to crack a program of average difficulty.

Within this tutorial, my definition of kracking is "changing the executable code of a program in order to change the behavior of the program."

The examples given in this tutorial are aimed at cracking application programs, but with the information given, it should be extendable to cracking any sort of computer software (i.e. startup extensions, control panels, etc.).

What Do You Need?

'Kay, you need some "can't live without" tools to crack any program. You'll need MacsBug, ResEdit, and (optionally, but recommended) the Code Editor for ResEdit. All of these can be found on the net; the Code Editor may be kinda tough to find, and some versions of SuperResEdit have it built in. The documentation that comes with all of these tells you how to install 'em.

Oh, and if you're like me, you'll also need a pencil and *lots* of paper.

MacsBug is a dissambler that lets you stop program execution at any time, at particular times, change anything in RAM, manipulate your computer's registers, alter program execution, cook toast, and lots of other good stuff. Much, much more on it later.

ResEdit is a resource editor. On the Smacintosh, files have two 'forks,' a data fork (for data) and a resource fork (for resources, duh). The data fork contains whatever the guy that wrote the program wants, while the resource fork contains 'chunks' of behaviors, styles, icons, cursors, fonts, or whatever. With ResEdit, you can easily change icons, fonts, cursors, the appearance of dialog windows, the appearance of alerts, patterns, or *EVEN* the code itself (Note: not on PowerSmac native apps).

Resources are specified by both a four character 'type' and a numerical ID. For example, the first code segment an application ever loads is specified by CODE ID 0. Here are some common types of resources: CODE
executable code for applications cdev
executable code for control panels INIT
executable code for extensions that run at startup CDEF
executable code that defines how a control (button, scroll bar, etc) behaves LDEF
executable code that defines how a graphical list of some sort behaves MDEF
executable code that defines how a menu behaves. WDEF
executable code that defines how a window behaves crsr
a color cursor CURS
a black & white cursor cicn
a color icon ICON
a black and white icon icl4
a 4 bit/pixel large color icon icl8
an 8 bit/pixel large color icon ICN#
a black & white large color icon ics4
a 4 bit/pixel small color icon ics8
an 8 bit/pixel small color icon ics#
a black & white small icon ppat
a color pattern ppt#
a collection of color patterns PAT#
a collection of black & white patterns sicn
a very small black & white icon snd
a sound STR
a string (collection of letters or numbers - a sentence) STR#
a collection of strings ALRT
a description of an alert window's placement & contents DLOG
a description of a dialog window's placement & contents MENU
a description of a menu's contents WIND
a description of a window's placement

This list is by no means comprehensive. Also, anything can be found in any kind of resource, if the guy that wrote the program is weird.

ResEdit allows you to edit any resource. When you open a file with ResEdit, you see a window containing a bunch of resource types. Double clicking on a resource type will show you another window containing all the resource IDs of

that type. Double clicking an ID will allow you to edit the resource with the selected type and ID. The editor window is different for different resource types. For example, when editing icons, cursors, or patterns, the window shows the resource graphically and contains rudimentary graphical editing/manipulation tools. When editing executable code, the window shows the hexadecimal version and the ascii version of the resource. When editing window descriptions, the editor shows a graphical version of the window. When ResEdit does not recognize the resource type, it reverts to the default hex/ascii view used by executable code resources.

Raw hex/ascii is not incredible useful unless you are a computer. A much nicer way of looking at executable code is in assembly language. The CodeEditor extension allows you to view resources in assembly within ResEdit. Whenever you edit a CODE, cdev, INIT, CDEF, MDEF or WDEF resource, instead of giving you raw hex and ascii, it dissassembles into assembly, lets you search for references to code snippits; basically, it's really cool.

If you have the CodeEditor extension, you can add additional resource types that it will edit. From ResEdit, open your ResEdit Preferences file (found in SystemFolder:Preferences) and add RMAP resources. If you've already installed CodeEditor, you can just check out any of the resources it edits (except CODE) to see how it's done.

**Programming Languages You'll Be Working With,

  and Ones You Won't**
(nice titles, huh...)

There are a few different types of language that you'll be dealing with here. There is assembly, which is a mnemonic language in which every instruction directly corresponds to something that your computer will do. There is machine language, which is the numerical equivalent of assembly language, and there are high level languages, which you won't have to deal with unless you're lucky enough to have the source code of the software you want to crack. Assembly language looks something like:

```
BNE.S      MYGETRESOURCE+00652
```

Machine language (for the same instruction) looks like:

```
6664
```

High level (for a different set of instructions) looks like:

```
if ((iAmKool && uAreNot) || (!iAmKool && uMightBe)) SomeonesKool();
```

The Toolbox Traps & MacsBug

All right, all I'm gonna cover is cracking programs that disable stuff or nag 'til you type in a registration code. Figuring out how to generate serial numbers is a lot tougher, 'cause you need a detailed knowledge of assembly.

Usually, you use the Macintosh Toolbox Traps to find out what's going on. The Toolbox is a set of routines that Smac programmers can use to simplify common tasks, making writing code really simple 'cause you don't have to do anything.

A trap is a system routine that performs some sort of action, such as drawing a menu bar or a window. Traps are stored within a program as a single instruction. When the trap is called, the program will perform the trap, then continue execution normally.

I'm going to cover the basic traps, but if you need a complete reference to all 5000+, you could check Apple's web site, follow the links to Developer pages, and get all the Inside Smacintosh books. You'll pretty much have to devote a hard drive to storing 'em on, but for basic cracking, you don't need 'em.

Here are some example program situations and the traps associated with them. These are *not* all associated with the nagging registration program. If the program puts up a window in which you have to click ok or cancel or whatever before you can do anything else, the odds are good that the trap used to create the window is GetNewDialog. The software will probably use the ModalDialog trap to automate handling events like mouse clicks and key hits. If the program is trying to get keystrokes at a weird time (like at system startup), it may use GetKeys. Close to the beginning of most application programs, the InitGraf trap will be called (this initializes some drawing variables). If the program puts up a window to tell you something while some other program is in the foreground (this is called a notification), it probably uses NMInstall. Programs have a main event loop that processes all

the mouse clicks, key strokes, etc. This loop will usually call WaitNextEvent, or, if it was written in 1910 (B.C.!), it may use GetNextEvent. To handle a menu selection, it will probably use MenuSelect.

'Kay, enough of this. If I haven't covered it, check Inside Smacintosh. Your trap will probably be in either Essential Toolbox or More Essential Toolbox. Check the chapter that seems relevant.

Allrighty, for our purposes we're going to assume that you want to crack a registration code, and the program puts up a window with Name, Organization, and Serial Number text boxes, has an Okay Button, and a Cancel button. Here's the basic strategy. You want to check out the code after you've filled in the three text fields and hit enter. You want to find where it determines whether your entry is valid, and make the program think that any entry is a valid one.

Now, you need to know a bit about MacsBug. MacsBug is a debugger for the Smacintosh; it allows you to examine code, memory, and even change things. Smac User Warning: MacsBug is *not* a pretty program. It takes over the entire screen. The majority of the screen displays whatever you tell it to (the main display area). At the bottom of the screen, it shows the next three assembly language instructions to be executed. At the very bottom of the screen is one line (the command line) where you can type commands. On the left side of the screen it displays (from top to bottom) the contents of your computer's stack, the name of the current running process, some environmental information, the state of the status register, the state of the eight data registers, and the state of the eight address registers.

You can do some pretty cool stuff with MacsBug, and if I don't cover it here, try typing ? on the command line for very good on-line help. The most important thing you can do is set a break point so that the program you are running will pause and you will drop into MacsBug on whatever toolbox trap you specify. This is an A-Trap Break. It uses the command atb <the trap name>. So, if you wanted to halt execution everytime the ModalDialog trap was found, you would use "atb modaldialog" (MacsBug is generally not case sensitive). You can clear an individual a-trap break using atc (a-trap clear). You can either use atc <the trap name> for an individual trap or atc to clear 'em all. By the way, using atb without a trap name will break on all traps, which I don't recommend unless you are clinically insane or chronically patient.

Anyway, at the bottom of the MacsBug screen, you will see a listing of (usually) 3 instructions. The current instruction is at the top, followed by the next two. The offset from the beginning of the procedure or resource in which the instruction resides is at the left, followed by the address in memory of the instruction, followed by the instructions mnemonic (the assembly language version), followed by the instructions arguments if any. On the right is the machine language version of the instruction. The machine language is in hexadecimal, and is what you would see if you opened a CODE resource in ResEdit without the CodeEditor. At the top of this listing is the name of the resource in which the code lives, or the name of the procedure. Finally, there may be more info following a semicolon. For example, if I go into MacsBug now, I get this listing at the bottom of the screen:

_SetResFileAttrs

```
; Will Loop
+006E2
4081B6DA
*DBEQ
D5,_SetResFileAttrs+006DE
; 4081B6D6
|57CD FFFA
+00636
4081B6DE
 BEQ.S
_SetResFileAttrs+006FE

; 4081B6F6
|6716
+006E8
4081B6E0
 BRA.S
_SetResFileAttrs+00704

; 4081B6FC
|601A
```

The name of the procedure (in this case, it's a toolbox trap) is SetResFileAttrs. The +006e2 is the offset from the beginning of the procedure. This instruction is 6E2 hexadicimal bytes from the beginning of the SetResFileAttrs trap. The 4081b6da is the actual address in memory of this instruction. DBEQ is the mnemonic of the instruction. D5,__SetResFileAttrs+006de is the instruction parameters. This instruction is used for looping. ;4081b6d6 tells what address it will go to if it loops. 57cd fffa is the machine language version of the instruction. The * in front of the mnemonic shows that it will be the next instruction to be executed. The Will Loop on the top line indicates that the instruction is going to loop. All instructions that conditionally jump elsewhere in memory will have something like this.

The next instruction shown is 636 hexadicimal bytes from the beginning of the SetResFileAttrs trap. It is located at address 4081B6DE in memory. It's mnemonic is BEQ.S. The paramters are _SetResFileAttrs+006FE. This instruction is a "Branch if Equal" (more later). If it branches, it will branch to 4081B6F6. It's machine language equivalent is 6716.

Aside - Ya Gotta Know Some Assembly Language

Using MacsBug is sort of pointless without at least a meager knowledge of assembly language. So, following are some of the assembly language commands important to cracking and finding your way around a program.

Bcc Instruction
Programs utilize conditional branches. This can be illustrative in a high level way by something like "if this is true go here, otherwise go over here." In assembly language, this is done with the mnemonic Bcc, where cc specifies what condition the statement will test. Some examples are BEQ (Branch if Equal), BNE (Branch if Not Equal), BGE (Branch if Greater than or Equal), BLE (Branch if Less than or Equal), BGT (Branch if Greater Than), and BLT (Branch if Less Than). There are a few more, but they aren't common. If a branch statement's condition is satisfied, the next instruction to be executed will be the instruction located at the address specified by this branch instruction's parameters, instead of being the next instruction in memory.

The various branch instructions test bits in the Status Register (SR - found in the middle of the left side of MacsBug). The bits tested depend on the branch instruction used. These bits are set by the instructions proceeding the branch instruction (more later). The state of the bits themselves is generally not relevant to kracking stuff.

A conditional branch's mnemonic will always begin with a B and the machine language equivalent will always begin with a 6.

You will probably want to change branch behavior. If a branch is going to branch, you may want to see what happens if it doesn't. Often, this is all it takes to crack a program; *finding the right branch is the tough part*. If this is the current line in MacsBug:

_DeQueue

; Will Not Branch
+000A8
408099fE *BNE.S

```
_DeQueue+000CA

; 40809A20


|6620
blah
40809A00
blah
blah



; blahhhh


|uggg
```

The next instruction to be executed is A8 hexadicimal byes from the beginning of the DeQueue trap. It is located at address 408099FE in memory. It's mnemonic is BNE.S. It's parameters are _DeQueue+000CA. This instruction will "Branch if Not Equal". If it branches, it will branch to address 40809A20. It's machine language equivalent is 6620.

In this example, the instruction is not going to branch. If you want to see what happens if it branches, type "pc=40809a20". The pc is a special address register that contains the address of the next instruction to be executed. This command changes the pc to the address that it would be if the instruction had branched (40809A20). If this instruction *was* going to branch and you wanted to see what would happen if it didn't, you could use either "pc=40809a00" or "pc=pc+2". It's "pc=pc+2" because the given BNE instruction takes two bytes in memory. This can be seen by looking at the machine language instruction 6620. A byte is two hexadecimal digits, so 66 20 is two bytes. If the machine language had been 6600 ff9a, you would use pc=pc+4.

The various branch instructions are the 'big boys' of program cracking. If a program does something you don't like, like displaying a "Register Me, Fucker!" screen or pausing before quitting, changing how a branch executes will almost always override the offending behavior. Again, FINDING THE CORRECT BRANCH STATEMENT IS THE TOUGH PART!!!!

CMP Instruction
There is also a compare instruction. It's mnemonic is CMP. It will (suprise!) compare two values and set the status register's (SR) bits according to the result of the comparison. It is used to set stuff up for a conditional branch statement. It's form is cmp.b, cmp.w, or cmp.l, plus two parameters. The .b, .w, or .l corresponds to compare a byte, a word, or a long. A byte is two hexadecimal digits, a word is four, and a long is eight. The two parameters are the things to be compared. These can be numbers, addresses, the contents of addresses in memory, or a whole ton of other things. The compare instruction will almost always be followed by a conditional branch of the form Bcc (you just read about 'em unless you're skipping around like a moron).

TST Instruction
There is a similar assembly language instruction that compares a paramter to zero. This is the TST (TeST) instruction. It's form is TST.B, TST.W, TST.L, plus one parameter. See the compare instruction for an explanation of the .B, .W, and .L part. Again, this is almost always followed by a conditional branch of the form Bcc.

JSR and BSR Instructions
Assembly language provides a way for a program to use the same bit of code in multiple places. Code can jump to the repeated part, execute it, and then return. This is done with the JSR (Jump to SubRoutine) instruction and the BSR (Branch to SubRoutine) instruction. For our purposes, these instructions are the same. Note that BSR is *not* a conditional branch. All of the following info about the JSR instruction also applies to the BSR instruction.

The JSR instruction will branch to a subroutine, execute the subroutine, then return to the instruction after the JSR. It's syntax is JSR <address of the subroutine>. If you're lucky, in MacsBug the address may be replaced by the name of the subroutine, instead of being something cryptic. Unfortunately, this doesn't always happen.

RTS Instruction
The RTS (ReTurn from Subroutine) will return program execution to the instruction following the JSR or BSR that called the subroutine in which the RTS instruction is found. It takes no parameters, and is always the last instruction in a subroutine. Since my symantecs suck, here's a sort of flowchartie type thing on how this works.

program is executing Routine A

a JSR or BSR instruction is executed with a parameter of Routine B - the instruction after this one in memory is Instruction A

program is now executing Routine B

an RTS instruction is found

program execution continues in Routine A at Instruction A

MOVE Instruction
The MOVE instruction moves something from one address in memory to another. It's form is MOVE.B, MOVE.W, or MOVE.L, plus two parameters. This instruction is commonly used to make a copy of something, or to pop stuff onto or off of the stack before or after calling a subroutine. Most subroutines need some sort of data to work with, so the routine calling it needs to be able to

communicate this data to the subroutine. It can do this by pushing stuff onto a stack, where a stack is essentially just what it sounds like. The stack can be viewed at the top left of the MacsBug screen. The address register A7 always points to the bottom of the stack. The weird thing about this stack is that you don't push things onto the top of it. The top is fixed, and things are pushed onto the bottom. So, the stack grows downwards. Often, subroutines return some sort of data on the stack. After the subroutine has executed, this data can then be popped off of the stack for use by the calling routine. Here's an example of an assembly language program passing three parameters (Parm1 - longword, Parm2 - word, and Parm3 - byte) to a subroutine called IAmASubroutine, then copying the result (which is a byte) into a variable called Result. This is meant to be illustrative; in MacsBug, you won't see names like these, only weird looking stuff.

```
move.l
Parm1, -(A7)

move.w
Parm2, -(A7)

move.b
Parm3, -(A7)

jsr

IAmASubroutine

move.b
(A7)+, Result
```

All you really need to know is that -(A7) pops something onto the stack, while (A7)+ pulls something back off.

As an aside, if the second parameter of the MOVE instruction is a data register, the move instruction will also set the Status Register's (SR) bits so that a compare instruction is not necessary.

A common use of this in the Registration Code Example is passing the serial number you had typed to a subroutine that checks it. The subroutine then returns a "yes or no" byte. This is then checked. Here's what this would look like:

```
move.l
<Address holding your registration code>, -(A7)

move.l
<some other type of info to check it against>, -(A7)

jsr
```

```
CheckItOut

move.b
(A7)+, D0

bne

ItsGood

ItsNotGood here
```

This pops my registration code onto the stack, pops something else onto the stack, calls the CheckItOut subroutine, moves the result into data register 0, then branches only if the result is not zero.

NOP Instruction
If you want an instruction that doesn't do anything except waste space (and you actually may), you can use the NOP (No OPeration) instruction.

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 damn 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 "Hey, Dipshit! Wrong Code!" 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?

Changing The Program
(gawd, nice titles, huh...)

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 hell with 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 29…tical'+02B54
; 05E4B8A4

|671C
```