`Beta Notes: 10/17/91

The following bold entries constitute a tentative outline for topics to dicuss in detail. Some of these topics will require a fair amount of research on my part - in particular, the Eve and Encryption sections will take some time. After this section come the live cracks. These represent an attempt to take a novice cracker through every step of the cracking process detailing choices and decisions that I would make as I go and why I would make them.

Any feedback would be greatly appreciated - especially from any novice crackers who find parts of this document incomprehensible. Note that this is a rough draft - there are bound to be errors although hopefully no logical ones (just syntactical and/or spelling).

Determining where to start looking

1) Types of protection
    a) Serial number schemes
    b) Registration codes
    c) Network serial checks [AppleTalk driver stuff]
    d) Hardware plugs - see below
    e) Encryption - see below
    f) Time stamps
    g) Key disk

How to break into programs

1) Trap interrupts
    a) Dialog/Alert traps
    b) MenuSelect
    c) InitFonts etc.
2) Manual entrance of TMON [Good luck]
3) Automatic TMON entrance via code modification [_Debugger trap insertion]
    a) Determining an address with Nosy
    b) Determining an address from the Jump Table
    c)

Using TMON, Nosy, and ResEdit together

1) Determining address offsets
2) Nosy vs TMON
    a) Why Nosy "feels better"
    b) Why TMON is virtually omniscient

TMON Tricks

1) TMON tricks with register values, flags, and instruction modification
2) One step ModalDialog hassles [Serial number schemes]
3) TMON Pro shortcuts

Determining the type of crack to apply

1) Bypasses vs cracks
2) Finding the key code
3) Branch switching
    a) Mention something about branch op-codes - 2 and 4 byte instructions and offsets

4) Flag/variable modification
5) Code modification

Everything you always wanted to know about the CODE 0 Jump Table.

1) What it is and how it works
2) Locating an entry point
3) Modifications

Hardware plugs

1) General tips [Device Manager stuff]
2) Eve bullshit

Encrypted Code

Unless you are one hell of a genius at cryptology and have lots of time to kill, the encrypted CODE resources will have to be de-crypted and written back to the program. Here is why: to decrypt itself, a program will usually either take a known seed number and use it on each encrypted byte of the code or else it will start with some byte in the code and do a forward decrypt, i.e. the first byte decrypts the second byte, the new second byte decrypts the third byte, and so on. A simple method might be to have some code that looks like this:

```
                MOVE    #1000,D0
                LEA     encryptedshit,A0
                LEA     encryptedshit-1,A1
loop1           EOR.L   (A1)+,(A0)+
                DBRA    D0,loop1
encryptedshit   Here is where the encrypted gibberish begins.
```

This is a simple example, but note how it functions. D0 gets the number of longwords to decrypt, A0 is the destination (where the decrypted stuff will go - which is right back over the encrypted stuff) and A1 gets the decrypting key which is the long word that was previously decrypted. Then the code simply loops D0 times writing over the encrypted code with the decrypted code. After this code has finished, the program continues execution right where the encrypted (and now decrypted) code begins. Now cosider: somewhere in the encrypted stuff is the error check that you have to modify. This will be simple enough to locate assuming that you can run the decryption routine and then immediately regain control in TMON. The problem is that when you go to modify the error check so that it always passes, the modification screws up the decryption routine. This is because the decryption routine requires the exact original values to run properly since these values are the keys that the code uses. So a crack using traditional methods requires that you not only change the error branch, but that you also change every other encrypted value such that the decryption routine still runs properly - no small feat!

A much more feasable method would be to decrypt the code, make the necessary modifications to the error routine, and then disable the decryption routine (just branching around it would do) and writing the whole mess (un-encrypted) back to the original code resource.

So much for the theory, now if I could just crack one of these suckers...

Live Cracks

MultiClip 2.0

This program uses a network checking algorithm to determine whether multiple copies with the same serial number are currently running - if you don't use this program on a network, you will never see the error.

Step 1: Where to start looking.

There are actually several good places to begin looking for the protection (especially if you have already cracked it - but I will assume that you have not). First of all, since the program scans the network, it is probably using the _Open Trap somewhere early in its code to to access the Appletalk driver. Second, it displays an error dialog (or alert) so we could open it up in Resedit, find the error dialog (and note its ID # for later use) and then Nosy it and look at procedures that call ModalDialog or one of the Alert traps to try and find the one that displays the dialog with the proper ID #. Third, we could have TMON trap either 1) ModalDialog if it is a dialog or 2) StopAlert, CautionAlert or NoteAlert if it is an Alert and begin tracing from that point backwords. Fourth, we could just Nosy it and start from the top (the slow way).

Whenever a program displays an error dialog (not a serial number dialog which seems to be in vogue these days) I almost always find the ID # of the dialog or alert and begin looking at procs in Nosy, so let's start there. In Resedit, we note that it is Dialog (and not Alert) #128 that is the problem. On to Nosy. After Nosy analyzes the INIT resource, open up the Trap Refs List under the Display menu and scroll down to GetNewDialog. Here you will find two listings: ASKNAME and PUTREGISTERDLOG. Since there are only two we can quickly check them both out (if there were a bunch, I would probably try a different method). First let us look at ASKNAME - here is the listing down to the GetNewDialog:

```
  42BA:                                  QUAL    ASKNAME ; b# =184  s#1  =proc54

                               vdu_1      VEQU  -26
                               vdu_2      VEQU  -18
                               vdu_3      VEQU  -12
                               vdu_4      VEQU  -10
                               vdu_5      VEQU  -8
                               param1     VEQU  8
                               funRslt    VEQU  12
   42BA:                                  VEND

                             ;-refs - com_43    MYFILTERFORNAME

   42BA: 4E56 FFE6      'NV..'  ASKNAME  LINK    A6,#-$1A
   42BE: 48E7 0318      'H...'           MOVEM.L D6-D7/A3-A4,-(A7)
   42C2: 2C2E 0008      2000008          MOVE.L  param1(A6),D6
   42C6: 42A7           'B.'             CLR.L   -(A7)
   42C8: 4EBA E642      100290C          JSR     proc19
   42CC: 285F           '(_'             POP.L   A4
   42CE: 486E FFF8      200FFF8          PEA     vdu_5(A6)
   42D2: A874           '.t'             _GetPort ; (VAR port:GrafPtr)
   42D4: 42A7           'B.'             CLR.L   -(A7)
   42D6: 302C 001E      '0,..'           MOVE    30(A4),D0
   42DA: D07C 0014      '.|..'           ADD     #20,D0
   42DE: 3F00           '?.'             PUSH    D0
   42E0: 42A7           'B.'             CLR.L   -(A7)
   42E2: 70FF           'p.'             MOVEQ   #-1,D0
   42E4: 2F00           '/.'             PUSH.L  D0
    42E6: A97C               '.|'                 _GetNewDialog ; (DlgID:INTEGER; wStorage:Ptr;
behind:WindowPtr):DialogPtr
```

The first thing to do is to locate the _GetNewDialog and determine its associated parameters: actually all we care about is the first parameter, the ID #. Tracing backwords, we see that -1 is the third parm, 0 is the second parm, and 30(A4) + #20 (from the ADD #20,D0) is the first parm. Well, we have a problem here. Instead of a nice plain ID # being passed to GetNewDialog, the ID # is hidden on the stack frame somewhere. At this point it is best to mark this proc as indeterminite and go on to the next one. If we

must come back to this one then we will have to figure out if ID #128 is valid for this proc and go from there. So let us look at PUTREGISTERDLOG

```
 33AC:                                  QUAL    PUTREGISTERDLOG ; b# =141  s#1  =proc35

                              vdb_1     VEQU  -286
                              vdb_2     VEQU  -278
                              vdb_3     VEQU  -276
                              vdb_4     VEQU  -274
                              vdb_5     VEQU  -272
                              vdb_6     VEQU  -270
                              vdb_7     VEQU  -268
                              vdb_8     VEQU  -264
                              vdb_9     VEQU  -262
                              vdb_10    VEQU  -256
                              param1    VEQU  8
 33AC:                                  VEND

                              ;-refs - INIT1

                              PUTREGISTERDLOG
 33AC: 4E56 FEE2      'NV..'            LINK    A6,#-$11E
 33B0: 2F0C           '/.'              PUSH.L  A4
 33B2: 206E 0008      2000008           MOVEA.L param1(A6),A0
 33B6: 43EE FF00      200FF00           LEA     vdb_10(A6),A1
 33BA: 703F           'p?'              MOVEQ   #63,D0
 33BC: 22D8           '".'    ldb_1     MOVE.L  (A0)+,(A1)+
 33BE: 51C8 FFFC      10033BC           DBRA    D0,ldb_1
 33C2: 42A7           'B.'              CLR.L   -(A7)
 33C4: 3F3C 0080      '?<..'            PUSH    #128
 33C8: 42A7           'B.'              CLR.L   -(A7)
 33CA: 70FF           'p.'              MOVEQ   #-1,D0
 33CC: 2F00           '/.'              PUSH.L  D0
  33CE:  A97C                  '.|'                    _GetNewDialog  ;  (DlgID:INTEGER;  wStorage:Ptr;
behind:WindowPtr):DialogPtr
```

Once again, find the GetNewDialog and determine the parms. Here we have -1 for the third, 0 for the second, and lo and behold, 128 for the first. This is definately our procedure. Note that this is an extremely easy example as no attempt has been made to disguise the ID # - it is clearly 128, the value we have been looking for all along.

Determining how to implement the crack.

The obvious place to start looking is just before the error dialog has been loaded. Here is that section of code from the above procedure:

```
             LINK    A6,#-$11E
             PUSH.L  A4
             MOVEA.L param1(A6),A0
             LEA     vdb_10(A6),A1
             MOVEQ   #63,D0
      ldb 1  MOVE.L  (A0)+,(A1)+
             DBRA    D0,ldb_1          Next comes the code we just looked at
             CLR.L   -(A7)
             PUSH    #128
             CLR.L   -(A7)
             MOVEQ   #-1,D0
             PUSH.L  D0
             _GetNewDialog
```

As we look at this code, keep in mind what it is that we are looking for. We know that the program is capable of loading without this error, so somewhere it has to be checking the network and then either branching to the error code (if it detects a copy of itself) or else branching around the error code. So we need to find the branch that is causing this segment of code to execute. A quick scan of the code that precedes the error dialog code should reveal nothing of interest. A Link followed by a 63 word Move Loop - no branches of any consequence whatsoever. If you are wondering why we can immediately eliminate the DBRA D0,ldb1 (after all, it is a branch) then ask yourself this: 1st, where does the branch go? Answer: to the line above the branch instruction. 2nd, what (if any) conditions is it checking? Answer: it checks to see if D0 (an obvious loop counter in this case) is equal to zero. If the branch does not either 1) branch directly to the error code (in this case it would have to be branching to the CLR.L -(A7)) or 2) branch around the error code (somewhere after the GetNewDialog and the ensuing ModalDialog and probably even an ensuing DisposeDialog) then the branch is almost certainly a bad candidate. You particulaly should be able to immediately eliminate loop terminator branches like the one above.

Well, since we have eliminated the only branch in this procedure above the GetNewDialog, we will have to look elsewhere. The next obvious place to look is in the procedure that called this one. Again Nosy makes this a snap. Take a look at the line right above the code listing that read refs - INIT1. The refs line tells you every procedure that calls the one you are currently looking at. Luckily, there is only one, so let us look at it next. Since this is a long procedure, I am only listing the section that surrounds the JSR PUTREGISTERDLOG line. I should also mention that I am writing this with a copy that I cracked a while ago and in un-cracking it for this document, could not remember exactly what the changed code was. I will show you where your code listing might differ from mine below:

```
196: 4268 0004      'Bh..'          CLR     4(A0)
19A: 4228 0006      'B(..'          CLR.B   6(A0)
19E: 4228 0007      'B(..'          CLR.B   7(A0)
1A2: 43FA 036E      1000512         LEA     data2,A1    ; len= 1
1A6: 45E8 0009      'E...'          LEA     9(A0),A2
1AA: 4EBA 0392      100053E         JSR     proc2
1AE: 43FA 03A2      1000552         LEA     data4,A1    ; 'Multi'
1B2: 4EBA 038A      100053E         JSR     proc2
1B6: 43FA 03AC      1000564         LEA     data7,A1    ; len= 2
1BA: 4EBA 0382      100053E         JSR     proc2
1BE: 4A6E FFEC      200FFEC         TST     vab_2(A6)
1C2: 6756           100021A         BEQ.S   lab_13
1C4: 4FEF FFFE      'O...'          LEA     -2(A7),A7
1C8: 2F2E FFEE      200FFEE         PUSH.L  vab_3(A6)
1CC: 4EBA 2C88      1002E56         JSR     proc29
1D0: 301F           '0.'            POP     D0
1D2: 6646           100021A         BNE.S   lab_13
1D4: 4FEF FFCE      'O...'          LEA     -50(A7),A7
1D8: 204F           ' O'            MOVEA.L A7,A0
1DA: 317C FFF6 0018 '1|....'        MOVE    #$FFF6,ioCRefNum(A0)
1E0: 216E FFEE 001E 200FFEE         MOVE.L  vab_3(A6),ioSEBlkPtr(A0)
1E6: 317C 00FC 001A '1|....'        MOVE    #252,CSCode(A0)
1EC: A004           '..'            _Control ; (A0|IOPB:ParamBlockRec):D0\OSErr
1EE: 4FEF 0032      'O..2'          LEA     50(A7),A7
1F2: 206E FFEE      200FFEE         MOVEA.L vab_3(A6),A0
1F6: A01F           '..'            _DisposPtr ; (A0/p:Ptr)
1F8: 486D FFFC          -4          PEA     glob1(A5)
1FC: A86E           '.n'            _InitGraf ; (globalPtr:Ptr)
1FE: A8FE           '..'            _InitFonts
200: A912           '..'            _InitWindows
202: A9CC           '..'            _TeInit
204: 42A7           'B.'            CLR.L   -(A7)
206: A97B           '.{'            _InitDialogs ; (resumeProc:ProcPtr)
208: A850           '.P'            _InitCursor
20A: 42B8 0A6C         $A6C         CLR.L   DeskHook
20E: 487A 0302      1000512         PEA     data2       ; len= 1
```

```
212: 4EBA 3198      10033AC          JSR     PUTREGISTERDLOG
216: 4EFA 0316      100052E          JMP     com_2
21A: 4227           'B''    lab_13   CLR.B   -(A7)
21C: A99B           '..'             _SetResLoad ; (AutoLoad:BOOLEAN)
21E: 42A7           'B.'             CLR.L   -(A7)
220: 2F3C 4452 5652 '/<DRVR'         PUSH.L  #'DRVR'
226: 487A 2156      100237E          PEA     data35      ; len= 12
22A: A9A1           '..'             _GetNamedResource ; (theType:ResType; name:Str255):Handle
22C: 1F3C 0001      '.<..'           PUSH.B  #1
230: A99B           '..'             _SetResLoad ; (AutoLoad:BOOLEAN)
```

First off, we need to find the line that calls the error procedure we just finished looking at. In this case the line will be either JSR PUTREGISTERDLOG or BSR PURREGISTERDLOG. We find the correct line just above lab 13. Now, quickly note the structure we are dealing with: we have JSR PUTREGISTERDLOG (which does all the error dialog stuff) followed by a JMP instruction. So the program is leaving the main flow of control after doing the error dialog. This is important because we can see that logically, there should be a branch that skips this piece of code and continues on with lab 13. If we scan backwords from the JSR PUT... we see a bunch of Initialization traps preceded by some Moves - but then notice this code:

```
JSR     proc2
TST     vab_2(A6)
BEQ.S   lab_13
LEA     -2(A7),A7
PUSH.L  vab_3(A6)
JSR     proc29
POP     D0
BNE.S   lab_13
```

Here is where I forget what the original code looked like so your listing might say BEQ.S lab 13 (for the second branch that is). Anyways, this code looks really good since it branches around the error section. At this point, we might hazard a guess and simply make these Branch instructions always execute by changing them to BRA lab 13. This might be an incorrect crack since the program could be making other checks above this code - we can eliminate this chance by continuing scanning upwards looking for references to lab 13 until the beginning of this procedure. What I would do in a case like this is make a real fast check of about 50 or so lines of code above this looking for branches refering to lab 13. If I find one, modify it...if not, then make the crack and test it. If the crack fails, then I would know to keep looking.

A quick note: The flow of the program seems to suggest that merely changing the first branch from BEQ to BRA would suffice since this instruction always executes (it is not branched around anywhere) and once this instruction branches to lab 13 there would be no need to change the second branch. However, I am writing this having already cracked this program and the method I used was to change the second branch only. Since I know that this works and cannot test any other method (not having a network at my disposal), I will proceed in this manner. The would-be cracker could certainly try changing the first branch and it looks to me as if this would work.

So how is the crack applied? Well, in this case, it looks like the program branches to lab 13 only if the serial check is OK (i.e. there are no extra copies running on the network) so we need to to make this branch always execute. The easiest way to do this is to change the BNE.S lab 13 to BRA.S lab 13 - branch not equal turns into branch always. So, simply pop over to Resedit and open the proper resource (INIT in this case). To determine the ID of the resource, look at the top of the procedure window in Nosy. The first line will contain an s# followed by a number. This is the segment number or ID # of the resource (in this case it is obvious since there is only one INIT resource, but for CODE resources this is really handy). Once the resource is open (make sure you do not have the Resedit disassembler running - if you do, select Open Using Hex Editor from the Resource menu) scan down to the line that most closely

matches the line you want to modify - in this case our line is 1D2 so find line 1D0 in Resedit and look over 2 bytes. There should be the code 6646. Just click in front of the 66, backspace to delete it, and type 60 (You can find these op-code numbers in the Cracker's Guide Part 1). Now quit and save changes and the crack is complete.

Infini-d 1.1

This program uses the common serial number / personalize dialog scheme.

Step 1: Where to start looking.

We have two good options here: 1) Find the Dialog ID # in Resedit and use Nosy's Trap Refs List or 2) trap ModalDialog in TMON and start tracing from there. I tend to use the second method, usually because I can implement the crack on the fly in TMON and actually run the program. Then I go back later and figure out how do a full crack with Nosy. Note that withe the second method we do not have to go through every stupid dialog in the program. Rather we can simply find the unfriendly ModalDialog and let TMON tell us which code resource we are in.

First, drop into TMON and set a trap intercept for _ModalDialog then exit TMON and launch Infini-D. TMON will proceed to stop execution at the first ModalDialog trap. Since it is possible for a program to have ModalDialog traps before the one that actually does the serial number stuff my first step is to immediately exit TMON and keep track of how many ModalDialogs occur before the serial number dialog comes up. In this case it is the first ModalDialog, so I would have to then quit and start over, this time not exiting TMON when the trap occurs.

Once you are in TMON, open an Assembly window to (PC) to look at the code that is executing. I forget exactly, but essentially what you would see is the ModalDialog trap followed by a couple of meaningless instructions and an RTS. Since nothing happens after the ModalDialog, we would need to Step through the RTS to get back to the procedure that called this one.

I should make a quick note here: this technique of making an on the fly crack via TMON usually means that you are going to ruin the application, i.e. you are going to end up with a serialized program that no longer needs to be cracked. This is not a true crack, rather this is a bypass - once this is done, the program is personalized and ready to run; in a sense you are letting the program crack itself. If you wanted to make a true cracked copy, you would have to look at exactly which branches were modified in TMON and then go into Resedit and change the same instructions (with an un-serialized copy of the application).

OK, enough about that. Here is the code you would see:

```
PEA      $157A(A5)
MOVE.L  $000C(A6),-(A7)
_ModalDialog
UNLK     A6
RTS
```

Since the procedure ends right after the ModalDialog call, we need to step through the RTS to see what called this procedure...and here is that code:

```
001E50B4: LINK.W    A6,#$FFFE
001E50B8: PEA      `FFFE(A6)
001E50BC: CLR.L    -(A7)
001E50BE: JSR      $1572(A5)
001E50C2: ADDQ.L    #8,A7
001E50C4: CMPI.W    #$0001,`FFFE(A6)
001E50CA: BEQ.S    ^$001E50D8
001E50CC: CMPI.W   #$0002,`FFFE(A6)
```

```
001E50D2: BEQ.S    ^$001E50D8
001E50D4: MOVEQ    #$00,D0
001E50D6: BRA.S    ^$001E50DA
001E50D8: MOVEQ    #$01,D0
001E50DA: TST.W    D0
001E50DC: BEQ.S    ^$001E50B8
001E50DE: CMPI.W    #$0001,`FFFE(A6)
001E50E4: BNE.S    ^$001E50EA
001E50E6: MOVEQ    #$01,D0
001E50E8: BRA.S    ^$001E50EC
001E50EA: MOVEQ    #$00,D0
001E50EC: UNLK     A6
001E50EE: RTS
```

Well, there is quite a bit of comparing and branching going on here so we had better see if we can figure out what is happening. After the Link, the dialog handle is pushed on the stack, space for a return value (or maybe a parameter with value 0) is put on the stack and then the ModalDialog procedure is called. This is pretty standard. Next, the stack is restored to its original value and something is compared to 1, branch if so, then compare the same thing to 2 and branch if so. Notice an important thing here, namely that this procedure never calls GetDItem or GetIText nor does it call any more subroutines so this procedure cannot be the one that checks the serial number. So it is probably a safe bet that this procedure is testing to see what exactly the user did - hit OK? hit Cancel? Type in a keystroke? Assuming for the moment that this is the case, take a wild guess what the various dialog item numbers are? You guessed it...1 is the OK button, 2 is the Cancel button. Now look at the code and you can quickly see what is happening (still assuming our item number theory is correct). First, if the item number hit was one (OK button) then branch down, and put a 1 in D0. If the item number hit was 2 (Cancel button) then do the same thing. Otherwise put a zero in D0. Finally, TST D0 and if it was 0 (neither button hit) then loop back and call ModalDialog again. At this point the program knows one of the buttons was hit. So, if it was not the OK button, branch down and put 0 in D0 otherwise put a 1 in D0 (so that's Cancel = 0, OK = 1). When we look at the procedure that called this one, we know that D0 will tell that procedure what happened (either OK or Cancel).

Note that this is one of those problem ModalDialog calls that exits everytime you hit a keystroke so you cannot just type in your name and serial number, hit OK to get back to TMON, and crack the sucker. Rather you have to either 1) settle for only typing in one letter before you crack it or 2) set a breakpoint just past the part were it tests for the OK button being hit, clear the ModalDialog trace, and exit - TMON won't interrupt until you hit the OK button and the breakpoint is encountered.

Finally, here is the last piece of code - the procedure that called the above procedure:

```
001E4FBE: ADDQ.L  #6,A7
001E4FC0:JSR     ^$001E50B4
001E4FC4: MOVE.W  D0,`FFFE(A6)          Here is where we returned from the above procedure. 1 = OK, 0 = Cancel
001E4FC8: CMPI.W  #$0001,`FFFE(A6)
001E4FCE:BNE.S   ^$001E5012             Branch if Cancel hit
001E4FD0:PEA     `FEF8(A6)
001E4FD4: MOVE.W  #$000A,-(A7)
001E4FD8:JSR     ^$001E4F58
001E4FDC: ADDQ.L  #6,A7
001E4FDE:PEA     `FEF8(A6)
001E4FE2:JSR     ^$001E52AC
001E4FE6: ADDQ.L  #4,A7
001E4FE8: MOVE.W  D0,`FFFC(A6)
001E4FEC:TST.W   `FFFC(A6)
001E4FF0:BNE.S   ^$001E5012
001E4FF2: MOVE.W  #$0001,-(A7)
001E4FF6:CLR.W   -(A7)
001E4FF8: MOVE.W     #$0034,-(A7)
```

```
001E4FFC:JSR     $107A(A5)
001E5000: ADDQ.L  #6,A7
001E5002: MOVE.L  582(A5),-(A7)
001E5006: MOVE.W  #$000A,-(A7)
001E500A:CLR.W   -(A7)
001E500C: MOVE.W  #$7FFF,-(A7)
001E5010: SelIText
001E5012: CMPI.W  #$0001,`FFFE(A6)        True if OK was hit
001E5018:BNE.S   ^$001E5020
001E501A:TST.W   `FFFC(A6)          Unknown: returned value from JSR above
001E501E: BEQ.S  ^$001E4FC0
001E5020: CMPI.W  #$0001,`FFFE(A6)
001E5026:BNE.S   ^$001E5070
001E5028:PEA     `FF38(A6)
001E502C: MOVE.W  #$0006,-(A7)
001E5030:JSR     ^$001E4F58
001E5034: ADDQ.L  #6,A7
```

Well, there is a lot of crap here and if you decided to trace the two JSRs you would be in for a long ride. The first thing to try is to deduce what will happen based on what we already know - we know that if the wrong serial number is entered, the program will go back to ModalDialog to let you change it. So we need to find a branch that goes back above line 1E4FC0 (the ModalDialog JSR). If we can find that branch and avoid it, we should be safe. So we will start tracing down from where the program returned, not making any assumptions yet, but looking at where the branches go. Right away you will note two JSRs. Take a look at the parameters passed, and you will note the pair of PEA FEF8(A6) instructions. So this same piece of information is being passed to both subroutines - nothing to write home about, but interesting. The real key you should notice here is that there is a TST and BNE after the second subroutine. This is the first chance the program has to make any decisions (although what decisions we don't know). Let's assume this branch does not execute (you could assume either way and wind up with the answer) i.e. FFFC(A6) = 0 - some stuff happens that we don't care too much about yet, some text is selected, and the button is tested. If it was OK, the return value from the second JSR is TSTed and if it was zero (which we are assumming), branch back to 1E4FC0 - back to the ModalDialog JSR. So this route is incorrect. Going back, we now need to assume that the branch at line 1E4FF0 did execute. This time, we jump right to the button check, skip the branch since OK was hit, and again TST the return value from the second JSR. Since the branch executed, this value cannot be zero, so execution proceeds. Looking down a few lines we note that there does not seem to be any more branches back to the ModalDialog JSR so we can tentatively assume that this is the end of the protection.

To apply the crack immediately, just make sure that branch executes. You can do this by typing BRA right over the BNE in TMON. If, however, you want to make a cracked, unserialized copy (which you can then serialize with anything you like) you need to figure out where code will be in Resedit and change that BNE to BRA. Unlike the listings I have pasted into this document, TMON will tell you exactly where the code is in the file. Refer to the above section on TMON MacNosy and Resedit for details, but essentially just find the Code Resource ID # and the offset from the TMON listing. Then Exit TMON and let Infini-d cancel out. Next open it the proper code resource in Resedit, scan down to the proper offset, and find the BNE (which is 66 in hex) and change it to BRA (60 in hex). Save changes and you are set.

FrameMaker 3.0

Serial number dialog scheme again. This one, however, presents a slight variation - Nosy won't disassemble it properly. This means that you will have to do all your cracking from within TMON.

Step 1: Where to start looking.

The only choice we have is to break in via TMON. The simplest way to do this is to drop into TMON, set a Trace Interrupt for ModalDialog and Exit. Now launch Framemaker 3.0 and wait for TMON to break in

Here is the code you would see: (note that this listing is from TMON Pro - a TMON 2.8.x listing will be slightly different)

```
005B4F88:'CODE'®$0003ƒ$040C+$0284 PEA     $01AA(A5)
005B4F8C:'CODE'®$0003ƒ$040C+$0288 PEA     `FDEC(A6)
005B4F90:P 'CODE'®$0003ƒ$040C+$2… _ModalDialog
005B4F92:'CODE'®$0003ƒ$040C+$028E MOVE.W  `FDEC(A6),D0
005B4F96:'CODE'®$0003ƒ$040C+$0292 EXT.L   D0
005B4F98:'CODE'®$0003ƒ$040C+$0294 MOVEQ   #$01,D1
005B4F9A:'CODE'®$0003ƒ$040C+$0296 CMP.L   D0,D1
005B4F9C:'CODE'®$0003ƒ$040C+$0298 BNE     ^$005B50EA                    ;'CODE'®$0003ƒ$040C+$3E6
005B4FA0:* 'CODE'®$0003ƒ$040C+$2… CLR.W   `FDEC(A6)
005B4FA4:'CODE'®$0003ƒ$040C+$02A0 CLR.B   (A3)
005B4FA6:'CODE'®$0003ƒ$040C+$02A2 TST.L   `96FA(A5)
005B4FAA:'CODE'®$0003ƒ$040C+$02A6 BEQ.S   ^$005B4FC4                    ;'CODE'®$0003ƒ$040C+$2C0
005B4FAC:'CODE'®$0003ƒ$040C+$02A8 MOVE.L  A4,-(A7)
005B4FAE:'CODE'®$0003ƒ$040C+$02AA PEA     $3802                                    ;$000037D8+$2A
005B4FB2:'CODE'®$0003ƒ$040C+$02AE JSR     $1702(A5)
005B4FB6:'CODE'®$0003ƒ$040C+$02B2 MOVE.L  A3,-(A7)
005B4FB8:'CODE'®$0003ƒ$040C+$02B4 MOVE.L  A4,-(A7)
005B4FBA:'CODE'®$0003ƒ$040C+$02B6 JSR     $419A(A5)
005B4FBE:'CODE'®$0003ƒ$040C+$02BA LEA     $0010(A7),A7
005B4FC2:'CODE'®$0003ƒ$040C+$02BE BRA.S   ^$005B4FE8                    ;'CODE'®$0003ƒ$040C+$2E4
005B4FC4:'CODE'®$0003ƒ$040C+$02C0 MOVE.L  `FDE8(A6),-(A7)
005B4FC8:'CODE'®$0003ƒ$040C+$02C4 MOVEQ   #$05,D0
005B4FCA:'CODE'®$0003ƒ$040C+$02C6 MOVE.W  D0,-(A7)
005B4FCC:'CODE'®$0003ƒ$040C+$02C8 PEA     `FDEE(A6)
005B4FD0:'CODE'®$0003ƒ$040C+$02CC PEA     `FDF0(A6)
005B4FD4:'CODE'®$0003ƒ$040C+$02D0 PEA     `FDF4(A6)
005B4FD8:'CODE'®$0003ƒ$040C+$02D4 _GetDItem
005B4FDA:'CODE'®$0003ƒ$040C+$02D6 TST.L   `FDF0(A6)
005B4FDE:'CODE'®$0003ƒ$040C+$02DA BEQ.S   ^$005B4FE8                    ;'CODE'®$0003ƒ$040C+$2E4
005B4FE0:'CODE'®$0003ƒ$040C+$02DC MOVE.L  `FDF0(A6),-(A7)
005B4FE4:'CODE'®$0003ƒ$040C+$02E0 MOVE.L  A3,-(A7)
005B4FE6:'CODE'®$0003ƒ$040C+$02E2 _GetIText
005B4FE8:'CODE'®$0003ƒ$040C+$02E4 MOVE.L  A3,-(A7)
005B4FEA:'CODE'®$0003ƒ$040C+$02E6 JSR     ^$005B5670                    ;'CODE'®$0003ƒ$040C+$96C
005B4FEE:'CODE'®$0003ƒ$040C+$02EA TST.L   D0
005B4FF0:'CODE'®$0003ƒ$040C+$02EC ADDQ.L  #4,A7
005B4FF2:'CODE'®$0003ƒ$040C+$02EE BMI     ^$005B50C8                    ;'CODE'®$0003ƒ$040C+$3C4
005B4FF6:'CODE'®$0003ƒ$040C+$02F2 CMPI.L  #$00000005,D0
005B4FFC:'CODE'®$0003ƒ$040C+$02F8 BGT     ^$005B50C8                    ;'CODE'®$0003ƒ$040C+$3C4
005B5000:'CODE'®$0003ƒ$040C+$02FC ADD.L   D0,D0
005B5002:'CODE'®$0003ƒ$040C+$02FE MOVE.W  ^$005B500A(D0.L),D0                  ;'CODE'®$0003ƒ$040C+
$306
005B5006:'CODE'®$0003ƒ$040C+$0302 JMP     ^$005B5008(D0.W)              ;'CODE'®$0003ƒ$040C+$304
```

If you try to step through this and enter your name etc., you will find that ModalDialog is exiting after any keystroke. The way to get around this hassle is to get rid of the Trace Interrupt and set a breakpoint after the OK button is hit. How you ask? Well, take a look at the code that follows the ModalDialog. First, D0 gets the dialog item that was modified. Next D1 gets the value 1 and the two are compared. From Resedit, you can find the dialog item numbers for all the items and it turns out that item 1 is the OK button, and item 5 is the serial number - these are the two important ones since the program can't proceed until the

OK button is hit (we don't care about the cancel button being hit) and then the program must check the serial number. Following the compare, we note that if they are not equal (i.e. OK button not hit) then it goes off somewhere. The next instruction must be the one that executes after the user hits the OK button. So set your breakpoint at the line that reads CLR.W FDEC(A6) which is at address 5B4FA0 (this will vary) - and in fact you can see the asterisk in the listing denoting that I have done just that. Now exit, enter your name and company and serial number (keep typing anything until the OK button lights up) and hit OK. Now TMON breaks in again at the breakpoint. Now we can begin the crack.

Determining how to implement the crack.

Before you continue, think about what the program must do at this point if it wants to validate your serial number (here it helps to have read Inside Mac on dialogs). First the program must obtain a pointer to the dialog item #5 (the serial number field) and then it must obtain a pointer to the text contained in that item. Knowing this, you can just scan down until you see a GetDItem trap followed closely by a GetIText trap. After this last trap, the program can do its validation. Here is that piece of code:

```
MOVE.L  A3,-(A7)
_GetIText
MOVE.L  A3,-(A7)
JSR      ^$005B5670
TST.L    D0
ADDQ.L  #4,A7
BMI      ^$005B50C8
CMPI.L  #$00000005,D0
BGT      ^$005B50C8
ADD.L    D0,D0
MOVE.W  ^$005B500A(D0.L),D0
JMP      ^$005B5008(D0.W)
```

We can note that A3 is the pointer that will point to the text after the trap. Once A3 has the text, a subroutine is called and D0 is tested. At this point, we cannot be sure whether the branch executes if the serial passed or failed, so we had better take a quick look at the code at address 5B50C8. I am not going to show it here, but that code does some crap then calles ParamText and then a Dialog call so it is probably safe to guess that the branch above jumps to the error code.

With this assumption in mind, what can we do about it? An initial guess would be to just make that BMI either not execute or even better, make the BMI branch down to the ADD.L D0,D0. Unfortunately, if you look at the last two lines, you can see that D0 not only determines whether the code branches to the error routine, but is then used for a JMP instruction so we had better take care of D0. Let's take a quick look at that JSR up a few lines that sets D0 in the first place and remember, we are trying to figure out what D0 should be set to. Also remember that the branch is a BMI meaning that the error occurs if the high bit of D0 is set.

```
004B1508:'CODE'®$0003ƒ$04C8+$096C LINK.W  A6,#$FF00
004B150C:'CODE'®$0003ƒ$04C8+$0970 MOVEM.L A3/A4,-(A7)
004B1510:'CODE'®$0003ƒ$04C8+$0974 LEA     `FF00(A6),A4
004B1514:'CODE'®$0003ƒ$04C8+$0978 MOVEA.L $0008(A6),A3
004B1518:'CODE'®$0003ƒ$04C8+$097C MOVEQ   #$00,D0
004B151A:'CODE'®$0003ƒ$04C8+$097E MOVE.B  (A3),D0
004B151C:'CODE'®$0003ƒ$04C8+$0980 MOVEQ   #$06,D1
004B151E:'CODE'®$0003ƒ$04C8+$0982 CMP.L   D0,D1
004B1520:'CODE'®$0003ƒ$04C8+$0984 BLE.S   ^$004B1526                    ;'CODE'®$0003ƒ$04C8+
$98A
004B1522:'CODE'®$0003ƒ$04C8+$0986 MOVEQ   #`FF,D0
004B1524:'CODE'®$0003ƒ$04C8+$0988 BRA.S   ^$004B1592                    ;'CODE'®$0003ƒ$04C8+
$9F6
004B1526:'CODE'®$0003ƒ$04C8+$098A MOVEQ   #$00,D0
004B1528:'CODE'®$0003ƒ$04C8+$098C MOVE.B  (A3),D0
```

```
004B152A:'CODE'®$0003ƒ$04C8+$098E MOVEQ   #$28,D1                                  ;'('
004B152C:'CODE'®$0003ƒ$04C8+$0990 CMP.L   D0,D1
004B152E:'CODE'®$0003ƒ$04C8+$0992 BGE.S   ^$004B1534                    ;'CODE'®$0003ƒ$04C8+
$998
004B1530:'CODE'®$0003ƒ$04C8+$0994 MOVEQ   #`FF,D0
004B1532:'CODE'®$0003ƒ$04C8+$0996 BRA.S   ^$004B1592                    ;'CODE'®$0003ƒ$04C8+
$9F6
004B1534:'CODE'®$0003ƒ$04C8+$0998 MOVE.L  A4,-(A7)
004B1536:'CODE'®$0003ƒ$04C8+$099A PEA     $3802                                      ;
$000037D8+$2A
004B153A:'CODE'®$0003ƒ$04C8+$099E JSR     $1702(A5)
004B153E:'CODE'®$0003ƒ$04C8+$09A2 MOVE.L  A4,-(A7)
004B1540:'CODE'®$0003ƒ$04C8+$09A4 JSR     $0532(A5)
004B1544:'CODE'®$0003ƒ$04C8+$09A8 MOVE.L  A3,-(A7)
004B1546:'CODE'®$0003ƒ$04C8+$09AA MOVE.L  A4,-(A7)
004B1548:'CODE'®$0003ƒ$04C8+$09AC JSR     $0392(A5)
004B154C:'CODE'®$0003ƒ$04C8+$09B0 TST.L   D0
004B154E:'CODE'®$0003ƒ$04C8+$09B2 LEA     $0014(A7),A7
004B1552:'CODE'®$0003ƒ$04C8+$09B6 BEQ.S   ^$004B1558                    ;'CODE'®$0003ƒ$04C8+
$9BC
004B1554:'CODE'®$0003ƒ$04C8+$09B8 MOVEQ   #$05,D0
004B1556:'CODE'®$0003ƒ$04C8+$09BA BRA.S   ^$004B1592                    ;'CODE'®$0003ƒ$04C8+
$9F6
004B1558:'CODE'®$0003ƒ$04C8+$09BC MOVE.B  $0001(A3),D0
004B155C:'CODE'®$0003ƒ$04C8+$09C0 SUBI.B  #$30,D0
004B1560:'CODE'®$0003ƒ$04C8+$09C4 BCS.S   ^$004B1590                    ;'CODE'®$0003ƒ$04C8+
$9F4
004B1562:'CODE'®$0003ƒ$04C8+$09C6 CMPI.B  #$02,D0
004B1566:'CODE'®$0003ƒ$04C8+$09CA BHI.S   ^$004B1590                    ;'CODE'®$0003ƒ$04C8+
$9F4
004B1568:'CODE'®$0003ƒ$04C8+$09CC MOVEQ   #$00,D1
004B156A:'CODE'®$0003ƒ$04C8+$09CE MOVE.B  D0,D1
004B156C:'CODE'®$0003ƒ$04C8+$09D0 ADD.W   D1,D1
004B156E:'CODE'®$0003ƒ$04C8+$09D2 MOVE.W  ^$004B1576(D1.W),D1
          ;'CODE'®$0003ƒ$04C8+$9DA
004B1572:'CODE'®$0003ƒ$04C8+$09D6 JMP     ^$004B1574(D1.W)               ;'CODE'®$0003ƒ$04C8+
$9D8
```

There are no traps here to quickly tell us what is happening, but we can quickly look at the lines that affect D0. Basically, there are a bunch of interspersed MOVEQ instructions putting various values into D0. One of the values is $FF which (since the high bit of $FF is set - in fact, all the bits of $FF are set) must trigger the error in the previous procedure. Other values include 5 and 0. Right now, that is enough information to proceed with the previous procedure - if we need more in depth info, we can always come back. So we have the following code again:

```
MOVE.L  A3,-(A7)
JSR      ^$005B5670
TST.L    D0
ADDQ.L  #4,A7
BMI      ^$005B50C8
CMPI.L  #$00000005,D0
BGT      ^$005B50C8
ADD.L    D0,D0
MOVE.W  ^$005B500A(D0.L),D0
JMP      ^$005B5008(D0.W)
```

Once again, we have an initial BMI which tells us that $FF won't work for D0. We also have BGT after comparing D0 with 5 which branches to the error - so D0 must be between 0 and 5 (the other values we noted from the subroutine above). At this point, I would (and did) simply try inserting values into D0. I started with 5 and the program went into Demo mode - strike one. Next I tried 1 and some other error occured. Finally, I tried 0 and the program continued flawlessly.

So you are asking, how exactly might you go about inserting these values into D0? Consider: once D0 is set to the proper value, the two branches become meaningless since they would not execute anyways (they only execute if there is an error). This little tidbit tells us that we can safely overwrite these instructions with anything we like. So we have several free bytes to put our own code into (don't panic yet - this is pretty straightforward) and all our code has to do is set D0 to 0 then proceed. One quick note: Never Never Ever modify code that affects the stack. If you do, you can easily cause system errors later on down the road. In the above code, this translates into not changing the ADDQ.L #4,A7 (A7 is the stack pointer, remember?). So what is the easiest way to put 0 into D0? Use a MOVEQ instruction. This is particularly nice because you probably do not know the machine hex code for instructions (like me). But that subroutine we looked at before is chalk full of MOVEQ instructions. If you look, a MOVEQ 0 #0,D0 translates into 70 00. So far so good except that the stupid BMI is one of those 4 byte branches. So we still have two bytes left that will be garbage since we just changed the first two. This is an excellent candidate for a NOP instruction - a two byte instruction that does absolutely nothing. The code for this (from the Cracker's Guide Part 1) is 4E 71.

So, open a dump window to the PC and find the BMI (I think it is 68 00 00 D4 or something like that). Change the four values to 70 00 4E 71 and now the program loads D0 with the correct value and proceeds as if nothing had happened. Now you have the crack, but you want to make a cracked / un-serialized copy right? So, unstuff a fresh copy of the application, open it in Resedit, and open the proper CODE resource. To find the ID #, look back at the TMON listing. It says CODE 0003 plus some benutia about the File reference number and then +nnnn where nnnn is the offset from the beginning of the Code resource. There is all you need. Open CODE ID 3 and jump down to line 2E8 (since 2EE is our byte) and change the 68 00 00 D4 to 70 00 4E 71. Now run it and enter anything you like for the serial number.

QuickFormat 7.01

[due to burn-out, the final sections have not been written up]

```
    33E:                                QUAL    CHECKFOR ; b# =508  s#3  =proc196

                            ;-refs -  3/INITPROG

    33E: 4E56 FFE4      'NV..'  CHECKFOR LINK    A6,#-$1C
    342: 48E7 0108      'H...'           MOVEM.L D7/A4,-(A7)
    346: 594F           'YO'             SUBQ    #4,A7
    348: 2F3C 6465 6D6F '/<demo'         PUSH.L  #'demo'
    34E: 3F3C 0080      '?<..'           PUSH    #128
    352: A81F           '..'             _Get1Resource ; (theType:ResType; ID:INTEGER):Handle
    354: 285F           '(_'             POP.L   A4
    356: 200C           ' .'             MOVE.L  A4,D0
    358: 6656           30003B0          BNE.S   lih_2
    35A: 594F           'YO'             SUBQ    #4,A7
    35C: 7004           'p.'             MOVEQ   #4,D0
    35E: 2F00           '/.'             PUSH.L  D0
    360: 4EAD 0082      10005EA          JSR     NewHandle(A5)
    364: 285F           '(_'             POP.L   A4
    366: 2F0C           '/.'             PUSH.L  A4
    368: 4EAD 0092      1000614          JSR     HLock(A5)
    36C: 2054           ' T'             MOVEA.L (A4),A0
    36E: 20BC 000F 423F ' ...B?'         MOVE.L  #$F423F,(A0)
    374: 2F0C           '/.'             PUSH.L  A4
    376: 2F3C 6465 6D6F '/<demo'         PUSH.L  #'demo'
    37C: 3F3C 0080      '?<..'           PUSH    #128
    380: 487A 007C      30003FE          PEA     data209     ; len= 2
     384: A9AB              '..'               _AddResource ; (theResource:Handle; theType:ResType;
theID:INTEGER; name:Str255)
    386: 554F           'UO'             SUBQ    #2,A7
```

```
388: A9AF           '..'             _ResError ; :OSErr
38A: 4A5F           'J_'             TST     (A7)+
38C: 6714           30003A2          BEQ.S   lih_1
38E: 3F3C 008B      '?<..'           PUSH    #139
392: 1F3C 0001      '.<..'           PUSH.B  #1
396: 4EAD 0462      2000B7C          JSR     DOSTANDA(A5)
39A: 554F           'UO'             SUBQ    #2,A7
39C: A9AF           '..'             _ResError ; :OSErr
39E: 4EAD 0452      20009FE          JSR     DOERROR(A5)
3A2: 2F0C           '/.'    lih_1    PUSH.L  A4
3A4: A9AA           '..'             _ChangedResource ; (theResource:Handle)
3A6: 2F0C           '/.'             PUSH.L  A4
3A8: A9B0           '..'             _WriteResource ; (theResource:Handle)
3AA: 2F0C           '/.'             PUSH.L  A4
3AC: 4EAD 009A      100061E          JSR     HUnLock(A5)
3B0: 2F0C           '/.'    lih_2    PUSH.L  A4
3B2: 4EAD 0092      1000614          JSR     HLock(A5)
3B6: 2E3C 176F 7C4E '.<.o|N'         MOVE.L  #$176F7C4E,D7
3BC: 2054           ' T'             MOVEA.L (A4),A0
3BE: BE90           '..'             CMP.L   (A0),D7
3C0: 6606           30003C8          BNE.S   lih_3
3C2: 422D FDE2        -$21E          CLR.B   glob73(A5)
3C6: 6020           30003E8          BRA.S   lih_4
3C8: 554F           'UO'    lih_3    SUBQ    #2,A7
3CA: 2F07           '/.'             PUSH.L  D7
3CC: 4EBA FE68      3000236          JSR     DODEMODI
3D0: 1B5F FDE2        -$21E          POP.B   glob73(A5)
3D4: 102D FDE2        -$21E          MOVE.B  glob73(A5),D0
3D8: 5300           'S.'             SUBQ.B  #1,D0
3DA: 670C           30003E8          BEQ.S   lih_4
3DC: 2054           ' T'             MOVEA.L (A4),A0
3DE: 2087           ' .'             MOVE.L  D7,(A0)
3E0: 2F0C           '/.'             PUSH.L  A4
3E2: A9AA           '..'             _ChangedResource ; (theResource:Handle)
3E4: 2F0C           '/.'             PUSH.L  A4
3E6: A9B0           '..'             _WriteResource ; (theResource:Handle)
3E8: 2F0C           '/.'    lih_4    PUSH.L  A4
3EA: 4EAD 009A      100061E          JSR     HUnLock(A5)
3EE: 4CDF 1080      'L...'           MOVEM.L (A7)+,D7/A4
3F2: 4E5E           'N^'             UNLK    A6
3F4: 4E75           'Nu'             RTS
```

Finder 7 Menus

```
458:                                 QUAL    GETPASSW ; b# =31  s#1  =proc14

                           vap_1      VEQU  -288
                           vap_2      VEQU  -280
                           vap_3      VEQU  -276
                           vap_4      VEQU  -274
                           vap_5      VEQU  -272
458:                                 VEND

                         ;-refs - DOCOMMAN

458: 4E56 FED8      'NV..'  GETPASSW LINK    A6,#-$128
45C: 48E7 0018      'H...'           MOVEM.L A3-A4,-(A7)
460: 4A2D FEFE        -$102          TST.B   glob59(A5)
464: 670C           1000472          BEQ.S   lap_1
466: 487A 01B4      100061C          PEA     data23      ; 'Password has already
46A: 4EBA 139E      100180A          JSR     OUTPUTTE
46E: 6000 00A2      1000512          BRA     lap_5
472: 3F2D FEBA        -$146 lap_1    PUSH    glob28(A5)
```

```
    476: A998           '..'            _UseResFile ; (frefNum:RefNum)
    478: 594F           'YO'            SUBQ    #4,A7
    47A: 3F3C 0101      '?<..'          PUSH    #257
    47E: 42A7           'B.'            CLR.L   -(A7)
    480: 70FF           'p.'            MOVEQ   #-1,D0
    482: 2F00           '/.'            PUSH.L  D0
     484: A97C               '.|'                _GetNewDialog ; (DlgID:INTEGER; wStorage:Ptr;
behind:WindowPtr):DialogPtr
    486: 285F           '(_'            POP.L   A4
    488: 2F0C           '/.'            PUSH.L  A4
    48A: 3F3C 0002      '?<..'          PUSH    #2
    48E: 486E FEEC      200FEEC         PEA     vap_3(A6)
    492: 486E FEE8      200FEE8         PEA     vap_2(A6)
    496: 486E FEE0      200FEE0         PEA     vap_1(A6)
    49A: A98D           '..'           GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR kind:INTEGER;
VAR item:Handle; VAR box:Rect)
    49C: 42A7           'B.'    lap_2   CLR.L   -(A7)
    49E: 486E FEEE      200FEEE         PEA     vap_4(A6)
    4A2: A991           '..'            _ModalDialog ; (filterProc:ProcPtr; VAR itemHit:INTEGER)
    4A4: 0C6E 0001 FEEE 200FEEE         CMPI    #1,vap_4(A6)
    4AA: 66F0           100049C         BNE     lap_2
    4AC: 2F2E FEE8      200FEE8         PUSH.L  vap_2(A6)
    4B0: 486E FEF0      200FEF0         PEA     vap_5(A6)
    4B4: A990           '..'            _GetIText ; (item:Handle; VAR text:Str255)
    4B6: 487A 0152      100060A         PEA     data22      ; 'cc5187efH28b911af'
    4BA: 486E FEF0      200FEF0         PEA     vap_5(A6)
    4BE: 4EBA FC4A      100010A         JSR     proc6
    4C2: 6642           1000506         BNE.S   lap_3
    4C4: 594F           'YO'            SUBQ    #4,A7
    4C6: 486E FEF0      200FEF0         PEA     vap_5(A6)
    4CA: A906           '..'            _NewString ; (theString:Str255):StringHandle
    4CC: 265F           '&_'            POP.L   A3
    4CE: 2F0B           '/.'            PUSH.L  A3
    4D0: 2F3C 5354 5220 '/<STR '        PUSH.L  #'STR '
    4D6: 3F3C 0080      '?<..'          PUSH    #128
    4DA: 487A 012C      1000608         PEA     data21      ; len= 2
     4DE: A9AB              '..'                _AddResource ; (theResource:Handle; theType:ResType;
theID:INTEGER; name:Str255)
    4E0: 3F2D FEBA        -$146         PUSH    glob28(A5)
    4E4: A999           '..'            _UpdateResFile ; (frefNum:RefNum)
    4E6: 1B7C 0001 FEFE   -$102         MOVE.B  #1,glob59(A5)
    4EC: 487A 00C0      10005AE         PEA     data20      ; 'Thanks for registeri
    4F0: 4EBA 1318      100180A         JSR     OUTPUTTE
    4F4: 4A2D FEFE        -$102         TST.B   glob59(A5)
    4F8: 6714           100050E         BEQ.S   lap_4
    4FA: 2F2D FEB0        -$150         PUSH.L  glob25(A5)
    4FE: 487A 0086      1000586         PEA     data19      ; ''Thank you for payin
    502: A91A           '..'            _SetWTitle ; (theWindow:WindowPtr; title:Str255)
    504: 6008           100050E         BRA.S   lap_4
    506: 487A 001A      1000522 lap_3   PEA     data18      ; 'For only $10, you ca
    50A: 4EBA 12FE      100180A         JSR     OUTPUTTE
    50E: 2F0C           '/.'    lap_4   PUSH.L  A4
    510: A982           '..'            _CloseDialog ; (dlg:DialogPtr)
    512: 4CDF 1800      'L...'  lap_5   MOVEM.L (A7)+,A3-A4
    516: 4E5E           'N^'            UNLK    A6
    518: 4E75           'Nu'            RTS
```