How to Crack: MAC [by The Vassal]

Hopefully this will be an informative work. One in which I will attempt to instruct you all how to successfully remove some of the more basic Mac protection schemes easily, the right way. Before we start anything, however, there are a few things that you will need. They are:

* Mac Plus or better w/1 meg of RAM suggested
* MacsBug (Debugger from APDA)
* Programmer's Key (Can invoke MacsBug from Keyboard)
* DisAsm 3.1 (Shareware disassembler)
* FEdit 3.21 (Shareware Sector editor)

...and

* A working knowledge of 68000 Assembler

Ok, whether or not you believe in Shareware, it is a good system and the programs mentioned above are all capable of being commercial packages, so please pay the tiny shareware fee so the authors will continue to make them better. Thanks.

Chapter 1- It's a Scheme Game

The first thing you should do before you crack any ware is to identify completely what scheme they are using to protect their ware. On the Mac, schemes are usually basic and easy to get around. Some of the more common ones include:

* Password Protection
* Serial Number Protection,
* Key Disk Protection
* Date Expiration Protection (for Betas usually)
* HardWare Key Protection

We'll look at these schemes one at a time and I will provide examples for each later. Right now, I want to explain how to determine what protection is what. First off, before you do anything make a backup of the application you are going to tamper with. This way if you screw up you will have something to go back to later.

Ok, now when you launch the application a couple of different protection schemes will surface immediatly. If a dialog comes up immediatly and asks you to type something in, it is almost always a Password Protection Scheme or a Serial Number Scheme. Key Disk Protection Schemes will ask you to insert a floppy so it can read important data from it and then continue with the program as normal. The Date Expiration Protection Scheme is never really noticed until it expires, then everytime you run the program it will tell you it has expired and will then

quit.

Probably the most overated protection scheme going is that of hardware 'key' or
'dangle' protection. Usually the software package would ship with a hardware device that you would connect to the ADB port, Serial Port, or SCSI port. The different methods of checking the hardware will be described in detail later on
in this document. No need to swamp you with technical stuff just yet.

Chapter 2- The Tools of Cracking

As I mentioned earlier you will need certain programs to help you along in the deprotection process.

MacsBug: Is a full-featured debugger that allows you to set traps in programs and then trace through instruction by instruction. This is an immeasurably useful program. It has loads of commands, but I only use these commands for cracking:

```
atb [trap]            ; lets you set a trap that which will break you into MacsBug if
                          the program tries to exectute it. Mac Traps are from A000-
                          AFFF and do many different things like _Eject & _ExitToShell
                          and stuff like that.
atc                   ; this will clear all of the traps t hat you set with atb
es                    ; quit current application and exit to shell
ea                    ; quit current application then launch it again
 G                    ; go. Continue the application as normal. Turn off MacsBug.
GT [addr]             ; lets you GO from a selected address.
il [addr] n           ; lists from selected address "n"= number of lines
?                     ; This displays the online help file, very useful
```

These are the commands I use most of the time. There are other commands which are
more complicated and do some special things but there is no need to explain them
here. I will do that in a future issue.

FEdit 3.21: This is a very good sector/file editor with good search functions for
finding certain code and changing it. This is very much like any other editor so
there is no need to explain its functions.

DisAsm 3.1: This is a disassembler, the only one I have seen on the Mac so far and it works pretty good. All functions are operated from the Menus, but the main
ones I use are the Search functions. Like finding certain traps, are certain addresses. I don’t really use this much but if needed it is good to have a

disassembler around. Sometimes MacsBug won't quite work if a program steals memory away from it and DisAsm must be used as a last resort.

Programmer's Key: This is a nifty little INIT that lets you invoke MacsBug from the keyboard. Basically you hit the Command-Reset keys and it dumps you into MacsBug, you can also hit Control-Command-Reset to restart your computer. Which is kind of neat. I recommend using this instead of the hardware interupt switch on the machine itself, mostly because its a pain to keep reaching in back of your machine to do it.

Chapter 3- Assembly is the key to the crack

It is extremely important to have a background in Assembly language if you want to get far cracking. To tell you the truth I used to be into programming the Apple II and IIgs in Assembly language. When I moved over to the Mac, I found out everything was C and Pascal, which disturbed me. C and Pascal allow you to program without having a clue what is really going on. I don't like the idea of it at all. Most people think cracking is something for people who know how to program, but the truth is, I haven't written a damn thing for the Mac as far as applications go, because I don't know how. I could probably write something in assembly, but I just don't have a nice assembler like Merlin on the Mac.

Some day, I recommend you buy a good 68000 reference manual so you can learn the processor. That's what I did but Apple II assembly knowledge helped me out alot.
Assembly languange on the Mac is a bit more complicated than that of the Apple II; on the Mac memory is moved not loaded or stored.

On the Apple II and IIgs there are three registers you would normally use to store data in. The A or Accumulator, the X Register, and the Y Register. The Mac has 16 registers that can be used in this manner. They are D0-D7 which are 8 data registers for storing data, they are all capable of holding 32-bits worth of data. You can access or change these the low 8-bits, the low 16-bits or all 32-bits, changing the low byte or word has no affect on the remaining unchanged portion. There are also 8 address registers from A0-A7 but A7 is usually used as a stack pointer for the 68020 and is also known as the SP in this situation. The address registers are basically the same as the data registers but they can only be accessed using all 32-bits. Changing the low word of an address register replicates the bit 15 in bits 16-31. This is called sign extension, which converts a two's complement 16-bit quantity into an equivalent 32-bit quantity.

Another register on the Mac worth noting is the program counter (PC) register. This holds the address of the next instruction to be executed, and is very useful
in tracing code.

There are many more opcodes in the 68000 instruction set then there are in the 6502 or 65c816, too many to list here. In the crack examples I will explain everything very thoroughly so you can get an idea of what is going on, and understand it.

MacsBug and DisAsm both list Assembler in the same manner. They list the address
on the left hand side, the opcode and effected address or values in the middle and the hex values on the right hand side. This is how I will lit my code, but I
will add in descriptions on what it is the code is actually doing. Here is an example of what I mean:

```
611F04:   Move.L    D0,D1     ;2200     : this moves the contents of
                                                    D0 into D1
```

That's just an example, but it is very common to see code like that. The thing that is cool about MacsBug is that it lists all of the Address and data registers
on the left part of the screen from top to bottom along with other system registers.

Another important aspect of 68000 and higher processors is that they use branches
the same way the Apple II did, only that there are many more different branch instructions and they are much more powerful. Here are a examples of branch opcodes for the 68000 series processors (used in Macs):

Hex Instr.	Ex.	Meaning
[$60]	$60xx	BRA (BRanch Always). This instruction always branches to an address $xx bytes in front of where the instruction was passed.
[$66]	$66xx	BNE (Branch if Not Equal). This instruction will branch ahead $xx bytes in front of where the instruction was just passed.

There are 16 in all but I would rather not make this publication into an Assembler Reference book. I will put out one of those in the future maybe. I do
suggest getting a 68000 quick reference though. On we go.

Chapter 4 - Key Code Protection

Example: Quick Format 7.0

The first type of protection scheme I would like to explain is the Key Code Scheme. This is used in Quick Format 7.0, which is a Public Domain Shareware program that is very good at formatting floppies and designing your own labeling scheme those floppies. It is definately worth the Shareware fee, so I suggest if you like the program to buy it, you can probably find it on many online services and from user groups.

The author decided to put in a registration algorithm which requires its users to type in a key code to access the advanced features of the program. When you first launch the Quick Format 7.0 application, a dialog box comes up and asks you to enter in the key code. If no key code is entered or you hit the return key, the program would continue to run, but with the advanced options turned off.

The author is doing a couple of different little things. First, he is going to check somewhere within his resource files to see if the current application being used is registered. Usually there is a register byte in a resource somewhere in the app. He will then do a compare to see if it really is registered. If it is, it continues like normal, if it isn't registered, it will jump to another routine which turns off the advanced options and then runs the app as normal.

There are a few options we have when deprotecting this app. We can use MacsBug to trace through for the routine, then disassemble it to see where it does the compare; or we can use ResEdit to find a resource that looks suspicious and delete it. The latter might be a little tedious and it is always much more interesting tracing through code.

Now we will deprotect Quick Format 7.0. For best results and for speed and memory purposes quit all other applications you are currently running. When you are back at the Finder, hit "Command-Reset" or reach in back of your machine and press the hardware interrupt switch, this will activate MacsBug. Your screen should have cleared and you should be looking at a white screen with numbers on the left hand quadrant of the screen running from TOP to BOTTOM. Those numbers are the various addresses and registers in memory.

Running along the bottom of the screen from LEFT to RIGHT are two separate boxes. The box on the big box with the numbers in it is a disassembly of the location

in
memory you just broke into with MacsBug. The smaller box under it is the MacsBug
command line. On the far left you should notice a blinking cursor. From the command line you can execute different commands to help you trace through programs, especially useful in deprotecting software. At the command line type "?" (help). This will print up a list of different topics. If you keep hitting return it will give you information about each topic in the order they are shown
on the screen. So play around and hit return a few times to get an idea of what
commands you can use.

Now that you are done playing let's get started. I almost always set a TRAP for
an _InitGraf. You can do this by typing

'ATB INITGRAF'

A message should appear above the disassembly box saying 'A-Trap Break at A86E (_InitGraf)' everytime. What this means is that the program will be stopped and
MacsBug will take over everytime the program tries to execute an _InitGraf. This
works the same way for all of the other trpas that the Mac toolbox has as well.

Ok, now type 'G' on the MacsBug command line. This should bring you right back to
the Finder where you started, and you will regain control of your Mac. Locate Quick Format 7.0 and launch it. Almost immediately your screen should change back
into the MacsBug screen. There should be a message saying 'A-Trap break at XXXXXXX : A86E (_InitGraf)'. This means when you launched the program, MacsBug halted it because the program tried to pass an _InitGraf trap. Now that the program is halted, you can TRACE through the program to find the copy protection.
You may not successfully pinpoint the protection to any one specific area until
you have traced through a number of times.

Use the 'T' command to trace through. The object is to continue hitting 'T' and
return until the protection scheme comes up. Eventually it will. When you do get
it up look at the last few lines of code that was passed and you should see something like this:

```
Addr            Instruction                    Hex Bytes
583834        JSR SETUPMEN                    4EBA FE14
583838        JSR INITIALI                    4EAD 02E2
58383C        JSR INITGLOB                    4EBA FEBE
583840        JSR VIRALCHE                    4EBA FF22
583844        JSR CHECKMOR                    4EBA F82A
```

Now, it's pretty obvious from just looking at the labels they used that you

can
determine what is going on. In most cases people would not use LABELS like the ones above, but since it is shareware and not a $500 commercial package I can see
why the author opted the easier route for programming ease. The first JSR would
probably be him initializing his menus and stuff. The second JSR would be to initialize the screen and the fonts or whatever, the third JSR would be initializing the global variables he would need and the fourth would be to check
for any virus, persay. The fifth however is the routine he uses to check if the
program has been registered and brings up the dialog asking you to enter a key code. If it hasn't been registered with the correct keycode the program turns off
some options. But, that is not necessary, as by omitting this JSR CHECKMOR you will remove the check and the program will run with all options available.

Write down the last 10 or so bytes on a piece of paper noting that 4EBA F82A is
what you will have to change. Since you want to omit these bytes you are best off
using two NOP (or No OPeration) commands. The hex value for a NOP is 4E71. Now run FEdit and open up the Quick Format 7.0 program and do a HEX SEARCH for the bytes you wrote down on the paper. Then change the proper values and you will be
all set. Here is what you should be looking for and the change you should be making:

Byte Changes (You should find the SEARCH string only ONE TIME!)

```
Search : 4EBA FE14 4EAD 02E2 4EBA FEBE 4EBA FF22 4EBA F82A
Change :                                         4E71 4E71
```

The protection showed above is obviously an easy scheme to get around, and to tell you the truth, there really aren't that many hard schemes on the Mac, like
there are or were on the Apple II. It is important to check the routine you are
disabling. Sometimes variables (or globals) are passed in between different parts
of protection schemes, if you skip the entire protection scheme there is a pretty
good chance you will miss a variable (or global) getting passed and your program
will crash on you in the future.

The best way to check is to use the program after you have initially deprotected
it, if it works ok, then chances are no globals were passed. In the example above, all of the globals were passed in the prior two JSRs, which made things very easy.

Chapter 5 - Serial Number Schemes

Serial Number Protection Schemes are the same as Password Protection Schemes in most cases. Both are checked with memory using a compare instruction, but they do have differences. For example, certain serial number schemes are actually mathematical answers where the application will perform some complex arithmetic equation using such information as your name or company name, if the equations solution matches the serial number you type in the program will continue on like normal.

Some serial number schemes are easy and do not use any arithmetic all, some check to see if you enter a prime number, etc. There are many ways. Probably the most common is; the company making the software will add a resource to the application housing the serial number. When the package first loads, it will ask you for your serial number then will do a compare with the value entered and the value stored in the serial number resource.

Chapter 6- Key Disk Protection Schemes

There are a couple of different ways commercial software authors implement Key Disk Protection. Key disk protection essentially is software that requires the original floppy diskettes to run correctly. These types of programs come with an installer for copying them onto a hard drive. Some packages offer three installs to a hard drive; if you need more copies, you have to use the original disk each time you run that program.

Essentially what is happening here is this. On the floppy disk there are files with their INVISIBLE bit set meaning they will not bee seen or accounted for in the volume info. Therefore, if you try to copy the program onto a hard drive the invisible files will not be copied and the copy will not work properly. If you just turn the INVISIBILITY bit off using ResEdit and then copy the files, the program will work. But, this is only for programs that do not always need the key disk. Other programs require the key disk every time you run the application. In those cases, the key disk has a purposely bad block. When you run the application, it will read from the floppy and test the block’s status. If it is bad it will continue on knowing it is the original disk, it the block is ok it means that the program has been copied onto another disk which is not the original or key disk. The best way to operate around this scheme is to trace through the program to where it actually reads from the floppy, then change the BRANCH condition after the COMPARE. This will fool the program into always

thinking that the block that is read is bad.

Chapter 7- Date Expiration Schemes

These schemes occur out of the blue without notice most of the time. Most apps that use this scheme are beta apps that are expected to be updated before too long or demos of games and such. There are only two ways to read the current date
or time on a Macintosh, one is to use the _GetDateTime trap the other is to use
the Macintosh Global variable memory address $020C at which the current number of
seconds that has elapsed since Midnight, January 1st is stored. Basically both routines are extremely similar and would probably look something like this:

_GetDateTime (Protection Method)

```
          _GetDateTime               (Get current # of secs since 12am
01/01/04)
          CMPI.L expiredate,D1 (Compare the date with expiration date)
          BLE        notexpired       (If less than, then it hasn't expired)
          JSR        EXPIRED!         (Otherwise show the dialog and bomb
out!)
```

Address $020C (Protection Method)

```
          CMPI.L expiredate,$020C (Compare the current date with expire date)
          BGT        EXPIRED              (If current date is > then expire
date,
show dialog
                                                        & BOMB!)
          ---------                      (otherwise it hasn't expired and
everything is
                                                         ok!)
```

That should have given you some idea what the protection schemes look like, they
vary of course but that is what they will normally be like. The simple way to disarm these routines is to change the Branch Condition, for instance in the first protection scheme shown above it would be extremely easy to change the BLE
to a BRA which will automatically branch to the “notexpired” portion of the program.

For the second routine has an easy way to eliminate the protection scheme. It would be to change the 'BGT EXPIRED' into two NOPs. Which will totally eliminate
the check and fool the program into thinking everything is ok.

Chapter 8- Hardware Key Protection

Not many people know too much about this protection scheme. But one obvious

observation is that companies are making there own devices and confused companies
with low marketshare are buying into there hardware protection scheme, which really is not that good to tell you the truth.

Basically what the Eve™ hardware key does is it hooks up to your ADB port and has a chip in it which can be read from. All of the packages I have seen that use
the Eve™ protection key come with an Eve INIT. Basically this INIT reads one or
more values from the Eve key and stores it somewhere in memory. The program that
will then do a Compare to the memory to see if the values match if they do then
the program will continue, however if there is a null value found the program will warn you that the Eve Key is not present and then quit. If the value is incorrect it will tell you that you either have the Eve hooked up incorrectly or
that you are using an Eve from a different software package.

The easiest way around the Eve Hardware key would be to change the Branch conditions of the checks it does 1) to locate Eve and 2) to compare the values.
Once you have changed these branch conditions the program will work as it would
have if the Eve protection had not been implemented.

From what I have heard some software locations or other important data to the program is stored in the Eve™ that may be needed upon run time. For example, if
a program is encrypted and the decryption routine is stored on the Eve™ key it would be almost impossible to deprotect the program without a valid Eve™ key to
work with.

So, I would just use MacsBug the way we have been and just trace through for the
checks. Eve™ will let you know when something is wrong through the use of dialogs, so tracing is pretty easy. Find those branches and change the conditions.

Chapter 9- The Ultimate Protection Scheme

This is a hotly debated topic. Most people would say that no protection scheme can go uncracked, but I beg to differ. I am sure I can stir something up that will boggle a few minds for at least 6 months of everyday tracing. Which will probably force that cracker to stop working on it.

In this day and age all software should be unprotected and should be available through the Shareware system. Corporate America is crippling our economy and the
only ones benefitting from it are the few 'elite' businessmen or businesswomen.

The Shareware system ensures that the people/programmers who put their time and
effort into their projects recieve all of the money they bring in. Thats what Corporate America should be like. Spread the wealth amongst everyone. And work for the money you make, don’t cheat anyone out of it.

'This article is written in memory of the old apple pirates who have paved the way for a new breed of "elite pirate" life form. Hadn't it been for their patience and instruction our generation of pirates would have died off long ago.
Now I pass onto you my knowledge and ideas in hope that you will all use it, share it, work together with it and be "elite" toward one another. '

The Vassal

Many thanks to: Hipcheck, Agent Orange, The Terrorist and HackMan

Also thanks to: Far Side, Jim Heebner, Zelig, ShadowMan, The Embalmer, Mr_T,
The Wahoo, Grimm, Enigmatic Simplicity, Gandalf Greyhame,
and (High Voltage for reminding me what the Apple II days were
really like)...