Cracking Guides

Cracklist Tutorial: The Amatuer Crackist Tutorial 1.3
Cracking 101:   Novice Guide

Cracklist Tutorial

VOL 1   NUM 1

The Amatuer Crackist Tutorial

Version 1.3

By

Specular Vision

Special Thanks to:
Mr. Transistor
Ironman
The Grand Elusion
Banzai Buckaroo

Another fine PTL Production

Call The Myth Inc. BBS

Table of Contents:

(Page Numbers will be aprox. until final version is finished)

```
Introduction:
-------------

Due to the current lack of Crackers, and also keeping in mind
the  time it took me to learn the basics of cracking,  I  de-
cided  to put this tutorial together.   I will  include  many
files which I have found helpful in my many cracking  endeav-
ors.   It also has comments that I have included to  make  it
easier to understand.



Comments Key:
-------------

Comments in the following material will be made by one of the
following  and the lines that enclose the comments  show  who
made the comment.

Specular Vision = -------------
Mr. Transistor  = +++++++++++++
Ironman         = |||||||||||||


Special thanks to Mr.  Transistor, for coming out of "Retire-
ment" to help compose this document.
```

Let's start with a simple introduction to patching a program using the DOS DEBUG program. The following article will introduce you to the basic ideas and concepts of looking for a certain area of a program and making a patch to it.

By: Charles Petzold / Specular Vision
Title: Case Study: A Colorful CLS

This article originally appeared in the Oct. 14,1986 Issue of PC Magazine (Vol 15. Num 17.). Written by Charles Petzold.

The hardest part of patching existing programs is determining where the patch should go. You really have to make an intelligent guess about the functioning of the program.

As an example, let's attempt to modify COMMAND.COM so that is colors the screen on a CLS command. As with any type of patch try it out on a copy and NOT the original.

First, think about what we should look for. CLS is different from all the other DOS internal Commands, It is the only internal command that does something to the screen other than just write to it with simple teletype output. CLS blanks the screen and homes the cursor. Since it can't do this through DOS Calls (unless ANSI.SYS is loaded), it is probably calling the BIOS Directly. The BIOS Interrupt 10h call controls the video, and so the CLS command probably uses several INT 10h instructions. The machine code for INT 10h is CD 10.

(While this same method will work under any version of PC-DOS, Version 2.0 and later, the addresses I'll be using are from PC-DOS 3.1. Other versions of PC-DOS(or MS-DOS) will have different addresses; you should be absolutely certain that you're using the correct addresses.)

Load COMMAND.COM into DEBUG:

```
               DEBUG COMMAND.COM
```

and do an R (Registers) command. The size of COMMAND.COM is in register CX. For DOS 3.1's COMMAND.COM, this value is 5AAA.

Now do Search command to look for the CD 10 bytes:

```
               S 100 L 5AAA CD 10
```

You'll get a list of six addresses, all clustered close to-

gether. The first one is 261D. You can now pick an address a little before that (to see what the first call is doing) and start disassembling:

```
               U 261B
```

The first INT 10 has AH set to 0F which is a Current Video State call. The code checks if the returned value of AL (Which is the video mode) is less than 3 or equal to 7. These are the text modes. If so, it branches to 262C. If not, it just resets the video mode with another INT 10 at address 2629.

At 262C, the code first sets the border black (the INT 10 at 2630), then does another Current Video State call (at 2634) to get the screen width in register AH. It uses information from this call to set DX equal to the bottom right row and column. It then clears the screen by scrolling the entire screen up with another INT 10 (at 2645), and then sets the cursor to the zeroth row and zeroth column with the final INT 10 (at 264D).

When it scrolls the whole screen, the zero value in AL actually means blank the screen, the value of BH is the attribute to be used on the blanked area. In an unmodified COMMAND.COM, BH is set to 7 (Which is white on black) by the following statement at address 2640:

```
               MOV  BX,0700
```

If you prefer a yellow-on-blue attribute (1E), you can change this line by going into Assemble mode by entering:

```
               A
```

then entering

```
               MOV  BX,1E00
```

and exiting Assemble mode by entering a blank line.

Now you can save the modified file:

```
               W
```

and quit DEBUG:

```
               Q
```

When you load the new version of COMMAND.COM (and you can do so without rebooting by just entering:

```
               COMMAND
```

on the DOS command level), a CLS will turn the screen blue and display characters as yellow.

If it doesn't or if anything you type shows up as white on black, that probably means you have ANSI.SYS loaded. If you use ANSI.SYS, you don't have to make this patch but can instead use the prompt command for coloring the screen.

END.

That was just one section of a very large article that helped me to get started. Next we'll look at two other articles, both written by Buckaroo Banzi. These two articles CRACK-1 and CRACK-2 give you an introduction to the different copy protection schemes used on IBM PC's, and how to find and bypass them.

By: Buckaroo Banzai
Title: Cracking On the IBM PC Part I

Introduction

For years, I have seen cracking tutorials for the APPLE computers, but never have I seen one for the PC. I have decided to try to write this series to help that pirate move up a level to a crackest.

In this part, I will cover what happens with INT 13 and how most copy protection schemes will use it. I strongly suggest a knowledge of Assembler (M/L) and how to use DEBUG. These will be an important figure in cracking anything.

INT-13 - An overview

Many copy protection schemes use the disk interrupt (INT-13). INT-13 is often use to either try to read in a illegally formatted track/sector or to write/format a track/sector that has been damaged in some way.

INT-13 is called like any normal interrupt with the assembler command INT 13 (CD 13). [AH] is used to select which command to be used, with most of the other registers used for data.

INT-13 Cracking College

Although, INT-13 is used in almost all protection schemes, the easiest to crack is the DOS file. Now the protected program might use INT-13 to load some other data from a normal track/sector on a disk, so it is important to determine which tracks/sectors are important to the protection scheme.   I have found the best way to do this is to use LOCKSMITH/pc (what, you don't have LS. Contact your local pirate for it.)

Use LS to analyze the diskette. Write down any track/sector that seems abnormal. These track are must likely are part of the protection routine.   Now, we must enter debug. Load in

the file execute a search for CD 13.   Record any address show.

If no address are picked up, this mean 1 or 2 things, the program is not copy protected (right...) or that the check is in an other part of the program not yet loaded.   The latter being a real hassle to find, so I'll cover it in part II. There is another choice.   The CD 13 might be hidden in self changing code.   Here is what a sector of hidden code might look like

```
-U CS:0000
1B00:0000 31DB     XOR    BX,BX
1B00:0002 8EDB     MOV    DS,BX
1B00:0004 BB0D00   MOV    BX,000D
1B00:0007 8A07     MOV    AL,[BX]
1B00:0009 3412     XOR    AL,12
1B00:000B 8807     MOV    [BX],AL
1B00:000D DF13            FIST   WORD...
```

In this section of code, [AL] is set to DF at location 1B00:0007.   When you XOR DF and 12, you would get a CD(hex) for the INT opcode which is placed right next to a 13 ie, giving you CD13 or INT-13.   This type of code can't and will not be found using debug's [S]earch command.

Finding Hidden INT-13s

The way I find best to find hidden INT-13s, is to use a program called PC-WATCH (TRAP13 works well also). This program traps the interrupts and will print where they were called from. Once running this, you can just disassemble around the address until you find code that look like it is setting up the disk interrupt.

An other way to decode the INT-13 is to use debug's [G]o command. Just set a breakpoint at the address give by PC-WATCH (both programs give the return address). Ie, -G CS:000F (see code above). When debug stops, you will have encoded not only the INT-13 but anything else leading up to it.

What to do once you find INT-13

Once you find the INT-13, the hard part for the most part is over. All that is left to do is to fool the computer in to thinking the protection has been found. To find out what the computer is looking for, examine the code right after the INT-13. Look for any branches having to do with the

CARRYFLAG or any CMP to the AH register. If a JNE or JC (etc) occurs, then [U]nassembe the address listed with the jump. If it is a CMP then just read on.

Here you must decide if the program was looking for a protected track or just a normal track. If it has a CMP AH,0 and it has read in a protected track, it can be assumed that it was looking to see if the program had successfully complete the READ/FORMAT of that track and that the disk had been copied thus JMPing back to DOS (usually). If this is the case, Just NOP the bytes for the CMP and the corresponding JMP.

If the program just checked for the carry flag to be set, and it isn't, then the program usually assumes that the disk has been copied. Examine the following code

```
      INT 13      <-- Read in the Sector
      JC 1B00     <-- Protection found
      INT 19      <-- Reboot
1B00  (rest of program)
```

The program carries out the INT and find an error (the illegally formatted sector) so the carry flag is set. The computer, at the next instruction, see that the carry flag is set and know that the protection has not been breached. In this case, to fool the computer, just change the "JC 1B00" to a "JMP 1B00" thus defeating the protection scheme.

NOTE: the PROTECTION ROUTINE might be found in more than just
      1 part of the program

Handling EXE files

As we all know, Debug can read .EXE files but cannot write them. To get around this, load and go about cracking the program as usual. When the protection scheme has been found and tested, record (use the debug [D]ump command) to save + & - 10 bytes of the code around the INT 13. Exit back to dos and rename the file to a .ZAP (any extension but .EXE will do) and reloading with debug. Search the program for the 20+ bytes surrounding the code and record the address found. Then just load this section and edit it like normal. Save the file and exit back to dos. Rename it back to the .EXE file and it should be cracked.

***NOTE: Sometimes you have to play around with it for a
          while to make it work.

DISK I/O (INT-13)

This interrupt uses the AH resister to select the function to be used. Here is a chart describing the interrupt.

```
AH=0    Reset Disk
AH=1    Read the Status of the Disk
        system in to AL

    AL          Error
  ----------------------------
    00   - Successful
    01   - Bad command given to INT
   *02   - Address mark not found
    03   - write attempted on write protected disk
   *04   - request sector not found
    08   - DMA overrun
    09   - attempt to cross DMA boundary
   *10   - bad CRC on disk read
    20   - controller has failed
    40   - seek operation failed
    80   - attachment failed
(* denotes most used in copy protection)
AH=2    Read Sectors

  input
     DL = Drive number (0-3)
     DH = Head number (0or1)
     CH = Track number
     CL = Sector number
     AL = # of sectors to read
  ES:BX = load address
  output
      AH =error number (see above)
          [Carry Flag Set]
      AL = # of sectors read

AH=3 Write (params. as above)
AH=4 Verify (params. as above -ES:BX)
AH=5 Format (params. as above -CL,AL
             ES:BX points to format
             Table)
```

For more information on INT-13 refer to appendix A.

END.

In part II, Buck cover's Calls to INT-13 and INT-13 that are located in different overlays of the program. This is a method that is used often.

Cracking Tutorial II.

By: Buckaroo Banzai
Title: Cracking On the IBM PC Part II

Introduction

OK guys, you now passed out of Copy Class 101 (dos files) and have this great new game with overlays. How do I crack this one. You scanned the entire .EXE file for the CD 13 and it's nowhere. Where can it be you ask yourself.

In part II, I'll cover cracking Overlays and the use of locksmith in cracking. If you haven't read part I, then I suggest you do so. The 2 files go together.

Looking for Overlays

So, you cant find CD 13 in the .EXE file, well, it can mean 4 things.

1: The .EXE (though it is mostly .COM) file is just a loader for the main file.

2: The .EXE file loads in an overlay.

3: The CD 13 is encrypted &/or hidden in the .EXE file.

4: Your looking at the WRONG file.

I won't discuss case 1 (or at least no here) because so many UNP files are devoted to PROLOCK and SOFTGUARD, if you can't figure it out with them, your stupid.

If you have case 3, use the technique in part I and restart from the beginning. And if you have case 4, shoot your self.

You know the program uses overlays but don't see and on disk? Try looking at the disk with good old Norton's. Any hidden files are probably the overlays. These are the ones we are after. If you still can't find them, use PC-WATCH (this program is a must!!! For all crackists. Traps ALL interrupts).

Using PC-Watch to Find Overlays

Start up PC-Watch and EXCLUDE everything in the left Col.. Search the right Col. until you find DOS21 - OpnFile and select it.

Now run the program to be cracked.
Play the game until the protection is checked.
Examine you PCWatch output to see what file was loaded right before it.
This probably is the one holding the check.
If not, go through all the files.

You Have Found the Overlays

Great, now just crack the overlay as if it was a DOS file. You don't need to worry about .EXE file, debug can write an overlay file. Part I explains the basics of cracking. I suggest that you keep a backup copy of the overlay so if you mess up, and you will, you can recover quickly. Ah, and you thought cracking with overlays was going to be hard.

Locksmith and Cracking

The copy/disk utility program Locksmith by AlphaLogic is a great tool in cracking. It's analyzing ability is great for determining what and where the protection is.

I find it useful, before I even start cracking, to analyze the protected disk to find and id it's protection. This helps in 2 ways. First, it helps you to know what to do in order to fake out the protection. Second, it helps you to find what the program is looking for.

I suggest that you get locksmith if you don't already have it. Check your local pirate board for the program. I also suggest getting PC-Watch and Norton Utilities 3.1.(Now 4.1) All of these program have many uses in the cracking world.

END.

OK, now let's put some of this information into practice by examining a few cracks of some common programs. First we'll look at a Crack for Mean-18 Golf by Accolade. Accolade has been one of those companies that has a fervent belief in Copy Protection.

Title: MEAN-18 UnProtect For CGA/EGA Version

This crack works by eliminating the code that tests for known bad sectors on the original diskette to see if it is the genuine article or an illegal copy. The code begins with an INT 13 (CD 13 HEX), a DOS BIOS disk service routine followed a few bytes later by another INT 13 instruction. The program then checks the returned value for the bit configuration that signifies the bad sectors and, if all is as expected, contin-ues on with program execution.

The code that needs to be patched is in the GOLF.EXE file and in the ARCH.EXE file. It is identical in both files and lies near the end of each file.

In the following steps, you'll locate the start of the test code and patch it by replacing it with NOP instructions (HEX 90). The method described uses the DOS DEBUG utility but Norton's Utility (NU) works too.

Copy all of the files from the MEAN-18 disk onto a fresh floppy using the DOS COPY command and place your original diskette out of harm's way.

Assuming DEBUG is in the A: drive and the floppy containing the files to be unlocked is in the B: drive , proceed as follows:

First REName the GOLF.EXE file so it has a different EXTension other than .EXE.

```
               REN GOLF.EXE GOLF.DEB
```

Next load the file GOLF.DEB into DEBUG and displays the "-" DEBUG prompt.

```
               A:> DEBUG B:GOLF.EXE
```

Search for the beginning of the code to be patched by typing:

```
               - S CS:100 FFFF CD 13
```

Searches the file for the two byte INT 13 instruction. If all goes well, two addresses should appear on the screen.

```
               XXXX:019C
               XXXX:01A8
```

XXXX indicates that the numbers preceeding the ":" vary from system to system but the numbers following the ":" are the same on all systems.

The next step is to use the "U" command as indicated to un-assemble a few bytes in order to verify your position in the file)

```
               - U CS:019C
```

(Un-assembles 32 bytes of code. Verify the following sequence of instructions:

```
               INT       13
               JB        01E9
               MOV       AL,[BX+01FF]
               PUSH      AX
               MOV       AX,0201
               INT       13
               POP       AX
               JB        01E9
               CMP       AL,F7
               JNZ       01B5
```

These are the instructions you'll be patching out in the following step)

```
               - A CS:019C
```

This command assembles the new instructions you enter at the keyboard into the addresses shown. Beginning at CS:019C, and for the next 21 bytes, ending with and including CS:01B0, enter the no op command "NOP" (90h) followed by a <return> or <enter>. Just hit <enter> at address XXXX:01B1 to end the assemble command.)

```
               XXXX:019C  NOP <enter>
               XXXX:019D  NOP <enter>
                          .
                          .
                          .
               XXXX:01AE  NOP <enter>
               XXXX:01AF  NOP <enter>
```

```
               XXXX:01B0  NOP <enter>
               XXXX:01B1 <enter>
```

This just wipes out the section of code containing the INT 13 check.

Now do a HEX dump and verify that bytes 019C through 01B0 have been set to 90 HEX.

```
               - D CS:019C
```

If they have, write the patched file to the disk as follows)

```
               - W
```

This writes the patched file back to the disk where it can be run by typing GOLF just as before but now, it can be run from any drive, including the hard drive)

Now just [Q]uit or exit back to DOS. This command can be executed at any "-" DEBUG prompt if you get lost. No modification will be made to the file on the disk until you issue the "W" command.

```
               - Q
```

The process is the same for the ARCH.EXE file but because it is a different length, the segment address, (XXXX part of the address), will be different. You should find the first INT 13 instruction at address XXXX:019C and the second one at XXXX:01A8 as before.

You will again be patching 21 bytes and you will start with 019C and end with 01B0 as before.   After doing the HEX dump starting at address 019C, you again write the file back to the disk with a "W" command then "Q" uit.

Norton's utilities can also be used to make this patch.   Begin by searcing the GOLF.EXE or ARCH.EXE files for the two byte combination CD 13 (remember to enter these as HEX bytes). Once located, change the 21 bytes, starting with the first "CD" byte, to 90 (a NOP instruction). As a check that you are in the right place, the byte sequence in both files is CD 13 72 49 8A 87 FF 01 50 B8 01 02 CD 13 58 72 3C 3C F7 75 04.   After modifying the bytes, write the modified file back to the disk. It can then be run from any drive.

END.

That was the first the tutorial cracks, here's another crack based on the same ideas but using Norton's Utilities instead. The following is an unprotect method for Eypx Submarine. Eypx is another one of those companies bent on protecting the world.

By: Assembler Magic
Title: EPYX Submarine Unprotect

You will only need to make one modification to the main executable program of Submarine, SUB.EXE. I will assume that your computer has a hard disk and that you have a path to DOS. It's time to fire up DEBUG as follows:

```
               DEBUG SUB.EXE<cr>
```

The computer should respond with a "-" prompt. Now look at the registers, just to make sure everything came up okay. Type the letter "R" immediately after the prompt.   The computer should respond with a few lines of info as follows:

```
AX=0000  BX=0001  CX=6103  DX=0000  SP=0080  BP=0000  SI=0000
DI=0000  DS=12CE ES=12CE SS=37B2 CS=27FC IP=0010 NV UP EI  PL
NZ NA PO NC
     27FC:0010 8CC0       MOV     AX,ES
-
```

Note the value of CS is "27FC".   That is the hexadecimal segment address for the beginning of the program code in your computer's memory. It is highly probable that the value you see for CS will differ from mine. Whatever it is, write it down. Also, the values you see for DS, ES and SS will almost certainly differ from mine and should not cause you concern. The other registers should show the same values mine do, and the flags should start with the same values.

Next, we will do a search for Interrupt 13's. These are BIOS (not DOS) Interrupts built into the program which are used to ensure that the original disk is being used to run the program. The whole key to this unprotect scheme is to bypass these Interrupts in the program code. The tricky part of this unprotect is to find them! They are not in the segment of program code starting at the value of CS equal to "27FC". They are closer to the beginning of the program in memory. Easy enough! Reset the value of CS to equal the value of DS as follows; type immediately after Debug's "-" prompt:

```
               RCS<cr>
```

Debug will prompt you for the new value of CS with:

```
               CS:27FC:
```

You respond by typing the value of DS you saw when you dumped the registers the first time. For example, I typed "12CE<cr>". The value you type will be different. Debug will again respond with the "-" prompt which means we are ready to do our search. Type in the following after the "-" prompt:

```
               S CS:0 FFFF CD 13<cr>
```

The computer should respond with three lines of information which are the addresses of the three Interrupt 13 calls built into the program. The first four digits are the segment address and will equal to the value of CS you have just set. The second four digits following the colon are the offset addresses which are of primary interest to us. On my machine they came back as follows:

```
               12CE:4307
               12CE:431F
               12CE:4335
```

The segment addresses will be identical and the three offset addresses should all be relatively close together. Now look at the first offset address. (As you can see, mine was "4307".) Write it down. Now we do a bit of Unassembly.

Type "U4307<cr>" which is the letter "U", followed immediately (with no blank spaces) by whatever your first offset address turned out to be, followed by a carriage return. If you are not familiar with unassembled machine code, it will look like lines of gibberish as follows:

```
               12CE:4307 CD13        INT      13
               12CE:4309 4F          DEC      DI
               12CE:430A 744C        JZ       4358
                         .
                         .
               12CE:431F CD13        INT      13
               12CE:4321 4F          DEC      DI
                         .
                         .
               12CE:4324 BF0400      MOV      DI,0004
               12CE:4326 B80102      MOV      AX,0201
```

In my computer, Unassemble will automatically output 16 lines of code to the screen. Yours may differ. Note, in the abbreviated list I have shown above, the addresses at the beginning of the two lines which contain the Interrupt 13's (INT 13) correspond to the first two addresses we found in our search. Now we continue the unassemble, and here comes

another tricky part. Just type in "U<cr>" after the "-" prompt.

You'll get sixteen more lines of code with the third Interrupt 13 on a line which begins with the address (CS):4335 if you have the same version of Submarine as I do. It's not terribly important to this exercise, but it will at least show you that things are proceeding okay. Now type in "U<cr>" again after the prompt. You are now looking for three key lines of code. On my program they appear as follows:

```
               12CE:4335 07          POP      ES
               12CE:4356 5D          POP      BP
               12CE:4357 CB          RETF
```

The true key is the instruction "POP ES". This instruction begins the normal return sequence after the program has executed its Interrupt 13 instructions and accompanying checks. If Debug on your machine prints fewer than 16 lines of code at a shot, you may have to type in "U" more than twice at the "-" to find these instructions. (If you haven't found any of this stuff, either get help on the use of Debug or go back to using your diskette version!) Write down the offset address of the "POP ES" instruction; the four digits following the colon, which in my example is "4354". You're well on your way now, so please persevere.

The next step is to modify the program to JUMP around the code which executes the Interrupt 13's and go immediately to the instruction which begins the normal return sequence (again, it's the "POP ES". Type in the following instructions carefully:

```
               A4307<cr>
```

This first bit tells Debug that new Assembler code will be inserted at the address of the first Interrupt 13. If your first Interrupt 13 is at an address other that "4307", use the correct address, not mine. The computer will prompt you with the address:

```
               12CE:4307
```

After which you will immediately type:

```
               JMP 4354<cr>
```

This instruction jumps the program immediately to the normal return code instructions. Again, at the risk of being redundant, if your "POP ES" instruction is at a different address, use that address, not "4354"!

The computer will prompt you with the address of the next in-

struction if all went well. MAKE SURE you just hit the carriage return at this point. Debug will then return the familiar "-" prompt.

Now it's time to examine your handiwork. Let's do the unassemble again starting at the address of what had been the first Interrupt 13 instruction, but which is now the Jump instruction. Type in "U4307<cr>" or "U" followed by the appropriate address and a carriage return. The first line beginning with the address should appear as follows:

```
               12CE:4307 EB4B        JMP      4354
```

The key here is the four bytes immediately following the address. In my example they are "EB4B". Yours may not be. But, they are VERY IMPORTANT because they represent the actual machine code which is the Jump instruction. WRITE THESE FOUR BYTES DOWN AND MAKE SURE THEY ARE CORRECT.

Now if you want to have some fun before we go on, reset register CS to its original value by first typing "RCS<cr>" at the "-" prompt. Then type in the original value of CS that I asked you to write down. Using my example, I typed "27FC<cr>". Next, you will type "G<cr>" after the "-" prompt which means GO! If all went well, SUB should run at this point. At least it will if you put all of the Submarine files onto the diskette or into the hard disk subdirectory where youre working. If it didn't run, you may have made an error. Check through what you have done.

Don't give up at this point if it does not run. Your version of Debug may simply have not tolerated our shenanigans. When you are done playing, quit Submarine ("Alt-Q<cr>") and type a "Q<cr>" after the Debug prompt "-" appears.

Now comes the tough part. I can't walk you through this phase in complete detail, because you may be using one of several programs available to modify the contents of SUB.EXE. Debug is not the way to go, because it can't write out .EXE files, only .COM files.

Note: Another method of doing this is to REName the SUB.EXE file so it has a different extension other than .EXE before you enter DEBUG. That way after you've made the change you can then [W]rite then changes out to the file right in DEBUG. Then one drawback is that you can't run the program in DEBUG once you've changed the name.

You have to get into your sector modification package (NORTON works good) and work on the SUB.EXE file on your new diskette or your hard disk. Remember, I warned you that doing this on your hard disk is dangerous if you are not fully aware of

what you are doing. So, IF YOU MESS UP, it's YOUR OWN FAULT!

You are looking for the first occurrence of an Interrupt 13 (the "CD 13") using the search facility in your program. If you don't have the ability to search for the two-byte hexadecimal code "CD 13" directly, then you will have to manually search.

Note: Norton 4.x now has a search utility. When you get to the point of typing in the search text, just press the TAB key, and you can type in the actual hexadecimal code "CD 13".

Start at the beginning of SUB.EXE and proceed. Again, you want to find the first of the three (first from the beginning of the program).

I will give you a hint. I found it in NORTON at location 4407 hexadecimal which is location 17,415 decimal in the SUB.EXE program file. DOS standard sectors are 512 decimal bytes. Replace the two bytes "CD 13" with the "EB 4B" or whatever your Jump instruction turned out to be. Write or save the modified file.

That's ALL there is to modifying SUB.EXE. You can go ahead and execute your program. If you have followed my instructions, it should run fine. Get help if it doesn't. Now, you should be all set. You can load onto your hard disk, if you haven't already. You can run it from a RAM disk using a BAT file if you really want it to hum. Or, if you have the facilities, you can copy it from 5-1/4" floppy to 3-1/2" diskette and run it on machines which accept that medium if you upgrade to a new computer.

END.

Now let's take a look at a newer crack on the program, Space Station Oblivion by Eypx. At a first [S]earch with Debug and Norton's Utility no CD 13's could be found, and yet it was using them... So a different approach had to be taken...

By: PTL
Title: Space Station Oblivion Crack

First of all, you must determine which file the INT 13's are in, in this case it had to be the file OBLIVION.EXE since it was the main program and probably contained the INT 13's. So then rename it to a different EXTension and load it into Debug.

Then do a [S]earch for INT 13's.

```
               -S 100 FFFF CD 13
```

Which will promptly turned up nothing. Hmmm...

Next you might decide that, maybe, the code was modifying itself. So quit from Debug and load up PC-Watch, include all the INT 13 Calls. For those of you not familiar with PC-Watch, it is a memory resident program that can be set to look for any type of BIOS call. When that call is made PC-Watch prints to the screen the contents of all the registers and the current memory location that the call was made from.

After PC-Watch is initialized, then run the OBLIVION.EXE file from the hard disk, leaving the floppy drive door open, and sure enough, when the red light comes on in the diskette drive, PC-Watch will report the address's of some INT 13 calls. Which you should then write down.

From there, quit the game, reboot, (To dump PC-Watch from memory) and load the OBLIVION.EXE into Debug and issue a [G]o command with a breakpoint. What address should you use for a breakpoint? You guessed it, the same address PC-Watch gives you.

Well, it locked up did'nt it? Which is quite common in this line of work so don't let that discourage you. So next re-loaded it into debug and this time [U]nassemble the address that you got from PC-Watch. But instead of finding the INT 13's you'll find harmless INT 21's.

Hmm... could it be that the program was converting the CD 21's to CD 13's during the run? Well, to test the idea assemble an INT 20 (Program Terminate) right after the first

INT 21. Then I run the program, and yes immediately after the red light comes on the drive, the program will terminate normally.

Then [U]nassemble that same area of memory, and low and behold, some of the INT 21's have magically turned into INT 13's. How clever...

So, then it is just a matter of locating the address of the routine that it jumped (JMP) to if the correct disk was found in drive A:. Once you have that address, just go to the start of all this nonsense and [A]ssemble a JMP XXXX command. Where XXXX was the address to jump to if the original disk was in drive A:.

Then just [W]rite the file back out to the disk and [Q]uit debug, and then REName the file back to OBLIVION.EXE afterwhich it should work fine.

END.

Chapter III Removing Doc Check Questions

A new fad has recently started up with software vendors, it involves the use of "Passwords" which are either stored in the documentation or are actually the documentation itself. Then when you reach a certain part of the program (Usually the beginning) the program will ask for the password and you have to look it up in the Docs before being allowed to continue. If the wrong password is entered, it will usually drop you to DOS or take you to a Demo version of the program.

This new form of copy protection is very annoying, but can usually be cracked without too much effort, and the files and the disk are usually in the standard DOS format. So now we'll take a look at cracking the Doc check questions.

First of all we'll crack the startup questions in   F-15 Strike Eagle by MicroProse.

By: JP ASP
Title: F-15 Unprotect

Make a copy of the original disk using the DOS DISKCOPY program.

```
               >DISKCOPY A: B:
```

Then insert the copy disk in the A drive and invoke DOS DEBUG.

```
               >DEBUG
```

Now we'll [F]ill an area of memory with nothing (00).

```
               -F CS:100 L FEFF 0
```

Next we will [L]oad into address CS:0100 the data that is on the A: disk (0) from sector 0 to sector 80.

```
               -l cs:100 0 0 80
```

Now lets [S]earch the data we loaded for the area where the copy protection routine is.

```
               -s cs:100 l feff FA EB FD
```

Then for each of the occurences listed, use the address DEBUG returned in the [E]nter command below.

```
               -e xxxx 90 90 90
```

Here's the part we are interested in, it's where you change all the autorization codes to a space.   Notice how you can use the [S]earch command to look for ASCII text.

```
               -s cs:100 l feff "CHIP"
```

Then for each occurance of "CHIP" use the address DEBUG returned in the [F]ill command below.

```
               -F XXXX L F 20
```

Write out the modified data

```
               -W CS:100 1 0 80
```

Quit DEBUG

```
               -Q
```

You should now be able to DISKCOPY and boot from all copies also just press the space bar when it ask for ANY authority code and then press "ENTER". Now there is no need to remember (or look up) any codes that are so finely tucked away in the manual!

END.

Here is a similar method that was used break the passwords in the program BATTLEHAWKS 1945 by Lucasfilms. However Norton Utilities is used to search for the passwords and change them.

By: PTL
Title: BATTLEHAWKS-1945 Doc Check Crack

In keeping in line with their previous programs, Lucasfilms has released yet another program which uses Doc Checks for its means of copy protection, Battlehawks 1942.

When you run this program, it first goes through a series of graphic displays, then it goes through a series of questions, asking what type of mission you want to fly, such as Training, Active Duty, or which side of the war you want to be on.

Then right before the simulation begins, it shows you a picture of a Japanese Zero and ask you for a password which you

are then supposed to get by looking up the picture of the Zero in the User Manual and typing the corresponding password in. After which it enters the simulation, in the event you enter the wrong password, it puts you into a training mission.

Removing the Doc Check in a program like this is usually pretty easy. The ideal way to do it is to remove the Doc Check routine itself, but if you don't have all day to debug and trace around the code this might not be the best way. For instance if you only have your lunch hour to work on it (Like I did), then you need to use the standard Q.D.C.R.S. (Quick Doc Check Removal System).

How do you do a QDCRS? Well first of all, play around with the program, find out what it will and will NOT accept as a password. Most programs will accept anything, but a few (Like Battlehawks) will only accept Alpha characters.

Once you've learned what it likes, make an educated guess as to what program the Doc Check routine is in. Then load that program into Norton's Utility (NU).

At this point, take a look at the passwords, and write down the most unusual one that you can find (I'll explain later). Now type that password in as the search string, and let NU search through the file until it finds the password. Now a couple of things can happen.

1. It only finds one occurrence
2. It finds more than one occurrence
3. It doesn't find any occurrence

In the event of case 2 then YOU have to determine where the passwords are stored, you can do this by opening your eyes and looking.

In the event of case 3, go to the kitchen and start a pot of coffee, then tell you wife to go to bed without you, because you have a "Special Project" that you have to finish tonight. And by the way, Good Luck. You'll need it.

Hopefully case 1 will occur, now you have to take a look at the data and ask yourself 2 questions:

1. Are all the passwords the same length?
2. Is there a set number of spaces between each password?
3. Does the next password always start a certain number of characters from the first character of the previous password?

If you can answer yes to any of the above questions, you in
luck. All you have to do is change the passwords to spaces

(If the program allows that, Battlehawks doesn't) or change
them to you favorite character. The letter X works good, it's
easy to type and easy to remember.

If you can't answer yes to any of the questions then you ei-
ther need to bypass the Doc Check routine itself or you need
to be adventurous and experiment. Battlehawks will not follow
any of the above patterns, and your quickly running out of
time, so you'll have to try something, fast...

So just wiped out all of the data area with X's, all the
passwords and associated "garbage" between them.   Then saved
the changes and drop out of NU and into BH. Then when it ask
for the password, just filed the area with X's. Next thing
you know, you'll be escorting a bombing run on a Japanese
carrier.

So, this one turned out to be fairly simple.   Where you may
run into trouble is on Doc Checks that use a graphic system,
such as Gunship by MicroProse. When it comes to this type of
Doc Check, you almost have to bypass the routine itself. And
again, a good way to do this is with setting break points and
using the trace option in Debug.

END.

---
That was the easy version Doc Check crack, however there a
"Better" way to crack Doc Checks, is to bypass the routine
completely so the user can just press enter and not worry
about spaces.   Let's take a lot at this method by looking at
a crack for the program, Yeager's Advanced Flight Trainer, by
Electronic Arts.
---

By: PTL
Title: Yeager's Advanced Flight Trainer

Chapter 5 Cracking Self Booters

Now we'll take a look at cracking self booters. A few companies have found this to be the best copy protection scheme for them, one of which is DataEast, makers of Ikari Warriors, Victory Road, Lock-On, Karnov, etc... This posses a special problem to the Amateur Cracker, since they seldom use standard DOS formats. So let's jump right in!

This is the area where a "Higher than Normal" knowledge of Assembly Language and DOS Diskette structures, so first of all, the Basic's.

The Disk's Physical Structure

Data is recorded on a disk in a series of concentric circles, called Tracks. Each track if further divided into segments, called Sectors. The standard double-density drives can record 40 tracks of data, while the new quad-density drives can record 80 tracks.

However, the location, size, and number of the sectors within a track are under software control. This is why the PC's diskettes are known as soft-sectored. The characteristics of a diskette's sectors (Their size, and the number per track) are set when each track is formatted. Disk Formatting can be done either by the operating system or by the ROM-BIOS format service. A lot of self booters and almost all forms of copy protection create unusual formats via the ROM-BIOS diskette services.

The 5 1/4-inch diskettes supported by the standard PC BIOS may have sectors that are 128,256,512, or 1,024 bytes in size. DOS, from versions 1.00 through 4.01 has consistently used sectors of 512 bytes, and it is quite possible that this will continue.

Here is a table displaying 6 of the most common disk formats:

Type	Sides	Sectors	Tracks	Size(bytes)

S-8	1	8	40	160K
D-8	2	8	40	320K
S-9	1	9	40	180K
D-9	2	9	40	360K
QD-9	2	9	80	720K
QD-15	2	15	80	1,200K

S - Single Density
D - Double Density
QD - Quad Density

Of all these basic formats, only two are in widespread use: S-8 and D-9. The newer Quad Density formats are for the 3 1/2" and 5 1/4" high density diskettes.

The Disk's Logical Structure

So, as we have already mentioned, the 5 1/4-inch diskette formats have 40 tracks, numbered from 0 (the outside track) through 39 (the inside track, closest to the center). On a double sided diskette, the two sides are numbered 0 and 1 (the two recording heads of a double-sided disk drive are also numbered 0 and 1).

The BIOS locates the sectors on a disk by a three-dimensional coordinate composed of a track number (also referred to as the cylinder number), a side number (also called the head number), and a sector number. DOS, on the other hand, locates information by sector number, and numbers the sectors sequentially from the outside to inside.

We can refer to particular sectors either by their three-dimensional coordinates or by their sequential order. All ROM-BIOS operations use the three-dimensional coordinates to locate a sector. All DOS operations and tools such as DEBUG use the DOS sequential notation.

The BASIC formula that converts the three-dimensional coordinates used by the ROM-BIOS to the sequential sector numbers used by DOS is as follows:

```
     DOS.SECTOR.NUMBER = (BIOS.SECTOR - 1) + DIOS.SIDE
       * SECTORS.PER.SIDE + BIOS.TRACK * SECTORS.PER.SIDE
       * SIDES.PER.DISK
```

And here are the formulas for converting sequential sector numbers to three-dimensional coordinates:

```
     BIOS.SECTOR = 1 + DOS.SECTOR.NUMBER MOD SECTORS.PER.SIDE
       BIOS.SIDE = (DOS.SECTOR.NUMBER \ SECTORS.PER.SIDE)
       MOD SIDE.PER.DISK
       BIOS.TRACK = DOS.SECTOR.NUMBER \ (SECTORS.PER.SIDE
```

* SIDES.PER.DISK)

(Note: For double-sided nine-sector diskettes, the PC's most common disk format, the value of SECTORS.PER.SIDE is 9 and the value of SIDES.PER.DISK is 2. Also note that sides and tracks are numbered differently in the ROM-BIOS numbering system: The sides and tracks are numbered from 0, but the sectors are numbered from 1.)

Diskette Space Allocation

The formatting process divides the sectors on a disk into four sections, for four different uses. The sections, in the order they are stored, are the boot record, the file allocation table (FAT), the directory, and the data space. The size of each section varies between formats, but the structure and the order of the sections don't vary.

The Boot Record:

This section is always a single sector located at sector 1 of track 0, side 0. The boot record contains, among other things, a short program to start the process of loading the operating system on it. All diskettes have the boot record on them even if they don't have the operating system. Asisde from the start-up program, the exact contents of the boot record vary from format to format.

The File Allocation Table:

The FAT follows the boot record, usually starting at sector 2 of track 0, side 0. The FAT contains the official record of the disk's format and maps out the location of the sectors used by the disk files. DOS uses the FAT to keep a record of the data-space usage. Each entry in the table contains a specific code to indicate what space is being used, what space is available, and what space is unusable (Due to defects on the disk).

The File Directory:

The file directory is the next item on the disk. It is used as a table of contents, identifying each file on the disk with a directory entry that contains several pieces of information, including the file's name and size. One part of the entry is a number that points to the first group of sectors used by the file (this number is also the first entry for this file in the FAT).

The Data Space:

Occupies the bulk of the diskette (from the directory through the last sector), is used to store data, while the other three sections are used to support the data space. Sectors in the data space are allocated to files on an as-needed basis, in units known as clusters. The clusters are one sector long and on double-sided diskettes, they are a pair of adjacent sectors.

(From here on I'll continue to describe the basics of DOS disk structures, and assembly language addressing technics.

Here is a simple routine to just make a backup copy of the Flight Simulator Version 1.0 by Microsoft. I know the latest version is 3.x but this version will serve the purpose of demonstrating how to access the data and program files of a selfbooter.

By: PTL
Title: Microsoft Flight Simulator 1.00 Unprotect

This procedure will NOT convert the Flight Simulator disk to files that can be loaded on a hard drive. But... it will read off the data from the original and put it onto another floppy. And this should give you an idea of how to read data directly from a disk and write it back out to another disk.

First of all take UNFORMATTED disk and place it in drive B:. This will be the target disk.

Now place your DOS disk (which has Debug) into drive A:, or just load Debug off you hard disk.

```
               A>DEBUG
```

Then we are going to enter (manually) a little program to load the FS files off the disk.

```
               -E CS:0000 B9 01 00 BA 01 00 BB 00
                          01 0E 07 06 1F 88 E8 53
                          5F AA 83 C7 03 81 FF 1C
                          01 76 F6 B8 08 05 CD 13
                          73 01 90 FE C5 80 FD 0C
                          76 E1 90 CD 20

               -E CS:0100 00 00 01 02 00 00 02 02 00 00 03 02
                          00 00 04 02 00 00 05 02 00 00 06 02
                          00 00 07 02 00 00 08 02
```

Next we'll [R]eset the IP Register by typing.

```
               -R IP
```

And then typing four zeros after the address prefix.

```
               xxxx:0000
```

Next insert the original Flight Simulator disk into drive A:
and we'll run our little loader.

```
               -G =CS:0000 CS:22 CS:2A
```

Now enter a new address to load from.

```
               -E CS:02 0E
               -E CS:27 19
```

And run the Loader again.

```
               -G =CS:0000 CS:22 CS:2A
```

New address

```
               -E CS:02 27
               -E CS:27 27
```

Run Loader

```
               -G =CS:0000 CS:22 CS:2A
```

Here we'll do some [L]oading directly from the disk our-
selves.

```
               -L DS:0000 0 0 40
```

And the in turn, write it back out to the B: (1) drive

```
               -W DS:0000 1 0 40
```

Etc...

```
               -L DS:0000 0 40 28
               -W DS:0000 1 70 30
               -L DS:0000 0 A0 30
               -W DS:0000 1 A0 30
               -L DS:0000 0 138 8
               -W DS:0000 1 138 8
```

When we are all through, [Q]uit from debug and you should
have a backup copy of the Flight Simulator.

```
               -Q
```

And that's all there is to it.

END.

Cracking 101

CRACKING 101 - 1990 edition

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ INTRODUCTION ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

by Buckaroo Banzai

A long time a go, in a galaxy far far away, a great adventure took ... What, oh sorry, wrong textfile.

Hello my children. Let me introduce myself, I am the great cracking guru BUCKAROO BANZAI (the original) and I'm back after a couple of years of hiding (from the Feds? from the IRS? No, from this girl MaryLou. Let me tell you, she could ... oh well let's get back to the textfile).

Let me tell you a little history about cracking on the IBMpc. It all started about 11 years ago with an apple IIe. See, I owned one and always wanted to learn how to crack (I was already a good pirate). Unfortunately, I just never could get the hang of it.

Well anyway, then I got my PC, and right away started to learn to program. Soon, I had pick up oh 4 languages one of which was assemble language. So I started down the long road to becoming a crackist.

But the road was hard since unlike the apple, there were NO textfiles on cracking the PC. Several unprotects, but nothing that really told you what to do. But thanks to some of the better known crackists of the day (Thanks SPI for the help) I got through.

It was at that point I decided to give something back. And thus, after a long (and I mean long) night of sex, drugs and rock and roll I started on my first cracking textfile. (Ok, so there really wasn't any drugs)

Since then, I have written about 10 different textfiles, 4 utilities and cracked several dozen programs. So, why the long pause, well I never really stopped cracking. I just basicly stuck to myself. I never released any of my cracks cause I was never first but several of my cracking programs (most known is SECTOR-C) reached the pirate world.

So, why am I back. 3 reasons. First is because now DOC CHECKS have taken over the scene and nobody has really written about them (plus I'm tired of seeing my old textfiles butchered in "CRACKING" mags). Second is because I have some free time, and third, because it was there.

It feels kinda funny. I have written this intro file several times, and the whole series has been rewritten. What started off as 4 simple textfiles has grown. I have givin up trying to write a book. What I'm doing is as a new game reaches me, I will crack it, and then tell how it was done, highlight the odd quirks about the crack.

I have also compiled a preaty good reference on INT 13h. I have included it with this series. And in the near future, I hope to release several utilities that I use to help me crack.

As of this writing, I have 2 actual lessons done, and 2 ready to be written. For the first 2 lessons I touch on both types on copy protection (On disk copy protection with I.B.M.'s DRAWING ASSISTANT and dos checks with EOA's ESCAPE FROM HELL). I still have to compose 2 more files, 1 more on each type (usings STAR CONTROL and CHAMBER OF THE SCI-MUNTANT PREISTEST). From there, who the hells knows.

So anyway, sit back, watch, listen, learn and if that doesn't work, kick a small kid in the head...

-Buckaroo Banzai
 -the cracking guru

CRACKING 101 - 1990 edition

Lession #1

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ CRACKING DOS Files ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

By Buckaroo Banzai

Today I'm here to about is cracking a dos format (either .EXE or .COM) file. This, in my mind is releativly the simplest (in theory although pratice might say otherwise) type of crack to do.

There are really 3 steps in cracking a dos file. Step 1, is finding where the protection routine is. How to go about it, well, there are several diffrent methods. Here are the steps that I often use.

First, I will run the program under PC-WATCH (PW) trapping INT13 all functions and INT21 functions 3Dh and 3Fh. Why trap the functions. This will give (hopefully) a starting place to look for the protection. Once you have set the breakpoints, press [F4] to execute and you will drop to dos. When you do, PW should display several calls to INT13. What closly at the CS:IP of these calls. Record it for later because these are calls from dos. We will uses this data to recognize what is a call to the protection and what is not.

Next, run the program to be cracked. As it executes, PC-WATCH will show what files are opened (including the file you just ran since DOS uses function 3Dh to open a file when it executes one) and what (and more improtantly WHERE) data is read to. Makes a list saying what data is read in from what file. Here is an example. Lets say you ran the program "XXX.COM". While running, "XXX.COM", you noticed that 2 other files "YYY.BIN" and "ZZZ.BIN" were also opened. So make a list like this:

```
    XXX.COM          YYY.BIN          ZZZ.BIN
    ÄÄÄÄÄÄÄ          ÄÄÄÄÄÄÄ          ÄÄÄÄÄÄÄ
```

Now, lets say that after "XXX.COM" was opened, PW showed that there were 2 reads from "XXX.COM" (the way to tell where the data is being read from is by checking the BX register on calls to 3Fh and the AX registers after calls to 3Dh. Yes, you should select both INPUT REGISTERS and OUTPUT REGISTERS from the PW menu) 1 at aaaa:bbbb and 1 as cccc:dddd. Right after "YYY.BIN" was opened, PW showed data was read to eeee:ffff and then after "ZZZ.BIN" was opened, data was written to gggg:hhhh and iiii:jjjj. Now, our list looks like this:

```
    XXX.COM          YYY.BIN          ZZZ.BIN
    ÄÄÄÄÄÄÄ          ÄÄÄÄÄÄÄ          ÄÄÄÄÄÄÄ
    aaaa:bbbb        eeee:ffff        gggg:hhhh
    cccc:dddd                         iiii:jjjj
```

What we have just created in a program load map. This map shows where to program to be cracked is loaded in memory. Next, scan though the calls to INT13. Look for either calls that return with errors, calls that have high values in the CH (> 28) or CL (> 9) registers, or calls not made by dos (those calls that have a CS:IP diffrent from the one we copied down before we executed the program). Now, look at the CS:IP of the call to INT13. Match the segment address against the program load map. If only 1 match occurs, then you now know what module the check is in so continue on to step #2. If more than 1 match occurs, check the offset (IP). Find the one that is closest to one of the write address's offset. Once you find a match, then go on to step #2. If no match occurs after both steps, it's time to track through the program.

Tracking your way though the program is a real bitch. I do not like to do because it can just take to long. But here is an overview on how it is done.

The object, is to keep narowing down calls until disk

access if found.   How to do this.   Well, load the program
under debug.   Keep tracing through the program in till a
"CALL" instruction is found.   Jot down you current IP and
PROCEED (using debug P command) over the instruction without
tracing in to it.   If you end up at the next instruction
without access that disk, then you have not found the routine
you are looking for so keep going.   Search for the next
"CALL" and then the next and then the next etc.   At some
point, when you proceed over a call, the disk will either
check protection or load in a new module.

How to tell the diffrence, well PW is still active and
will tell you if it was a call to INT13 or INT21 or BOTH.   If
it was the call to INT13 or a call to BOTH then you have
found a call to the protection routine (although the actual
call may be 100 levels deeper, you are on the right track).
Exit and restart but this time when you reach the call,
trace into it.   Now do the same process until you get to the
call to the next level, then again for the next, etc.
Finally you should find where it is.

But hopefully, you won't have to do that.   As I said, it
is very time consumming.   Hopefully, you will know which
module to look in.   If you do, here is how to find the call
to the protection.   First, try the simple search method.
Load up the module using DEBUG and simply type:

```
          S CS:100 FFFF CD 13   (use CS:100 if .EXE)
```

Debug will hopefully list 1 or more address.   If not,
try the same command only using CS:0000.   If again you are
not givin any address, you have some tricky debugging at hand
(an I suggest rereading the exercise in self-modifying code).

I will explain in detail how to find self-modifying code
later but for now, lets assume we have found the protection
routine.   Next, is to figure out what the copy protection is
trying to do.   First, look to the printout from PW.   Look
through it until you find the calls the INT13 from the
protection routine.   Look at the AH register.   If it is 00h
then the protection routine is probally reading in data from
the protected tracks.   If not, then the protection is simply
looking for some KEY (in other words a damaged track or
sector) that DOS canno't duplicate.

The second choice is much eaiser to defeat.   2 quick
methods to defeating this type.   First, you can fake the call
and simply set the registers.   Take the follow check to a
protection routine:

```
1:      mov AX,0201h        ; Read 1 sectors using int 13h
2:      mov CX,2909h        ; Track 29h sector 09h
3:      xor DX,DX           ; Drive 0, head 0
4:      int 13h             ; Read sector
5:      jnc Bad             ; If no error then it's a copy
6:      cmp AH,10h          ; Was it a CRC error
```

```
7:         jne Bad                 ; No, then it's a copy
8:         mov AX,0h                 ; clear error flag
9:         jmp Done                 ; we are done
Bad:       mov AX,1                  ; set error flag
Done:      ret                     ; we are done
```

What is the above code trying to do. Well, it's checking for a KEY on track 29h. That key is sector 09h. Normally sector 09h would not have an error. On a read to the original disk, after the int13 (line 4) is executed, the carry flag (CF) would be set. The jnc in line 5 would jump if CF is not set (indicating no error, which is bad since the original disk would have an error there). The next line checks AH to see if it is 10h. This is checking to see if the error was a Bad CRC on the read (the error that should be there). If it was not, then again it is not the protected disk. Only after both of thoses conditions are met, will the protection routine return a "GOOD" result.

The key here is the value returned in AX an possibly CF. When the disk is the original, AX would return the value of 0000h and CF = 1 but when it was a copy, it would return 0001h and CF = 0 or 1. Since on a bad return, CF can be 0 or 1 then it is preaty safe to assume CF is not used to signal the return. So what must we do to beat the protection routine, well, simply return from this CHECK with AX = 0000h. Simple. Just change line 1 to "mov AX,0000h" and line 2 to "RET". This will just bypass the check.

Now, this example is quite simple and would probally never be used in a REAL protection routine. I kept it simple to show the point, see the example on how to crack DRAWING ASSISTANT for a better example.

The second and more perferd method is to simply bypass the call to the protection routine and kill of the section of code that test for the check position. Take the following example:

```
10:        call 1                         ; call the first example
11:        cmp AX,0                           ; Was it the original
12:        jz   Good2                         ; Yes, then good
13:        ... BAD it was a copy     ; No, then bad
Good2:
```

The above example, when used with the last example show a typical call to a protection routine. The perfered method to crack this protection would not be to simply fake the return, but to remove the call to the protection. How to do it, simple. Just jump over the check. Change line 10 to "jmp Good2". This will bypass the protection routine.

Now, you might ask why would you want to take the extra step of finding the call to the protection routine rather than simply faking an int13 and returning with the proper registers set. 2 reasons. First, What if there wasn't enough room to setup the registers the way you needed them. Then you would have to take the extra step. Second, what if somewhere down the line, that routine is used for something else (like the int13 is modified into an int10 in a game). Since you have changed the bytes at that location, the modifying routine would create code that wasn't exepcted. But as always, if you can fake the return, and the program works, leave it. After all, not to many people go around look at other peoples cracks (do they???).

Now, what to do, if the program actually reads in important data from the disk. Well, there are 2 ways to go about this (possibly more). First, you could patch the program so that when it calls it's protection routine, it jumps to your user routine that opens a file and reads in the data to the right place. This method is preaty simple to add to a .COM file but a much harder to patch on to the back of a .EXE. I won't really go in to this method much more than to say use your brains. It's not a difficult concept.

The other method, is to create a LOADER or a TSR. I suggest creating an Interrupt Service Routine (ISR) that handles loading in the data. For example, let say you wrote a routine to read in the needed data for a file. It is not to difficult to convert that routine into an ISR. (For notes on ISR and TSRs, try reading The Waite Group's "MS-DOS PAPERS". It has one of the best sections on the subject).

Consider this following example:

```
A:          call 1             ; test protection
B:          jnc Good           ; was it successfull
C:          ... BAD load       ; no then it's a copy
            ... EXIT TO DOS    ; so exit to dos
Good:       ... Good load      ; yes then it original
            call 7C00:0000     ; then jump of protection
                               ; data

1:          mov ax,0209        ; Read 9 sectors starting from
2:          mov cx,290a        ; Track 29h Sector 0Ah (10)
3:          xor dx,dx          ; for drive A: head 0
4:          mov bx,7c00        ; read to 7c00:0
5:          mov es,bx          ;
6:          mov bx,0           ;
7:          int 13h            ;
8:          ret
```

What the above example dos. Lines 1-8 try to read in sectors 0Ah - 12h for track 29h on drive A:. This is the protection check routine. Lines A - Good attempt to check the protection, and then if the check is good (CF = 0) then a call to the loaded in code (the data loaded in by the protection check) is made.

What we want to do, is somehow when INT 13h is called, load in the needed data for disk.   Well, here is my suggestion.   First, I would change line 7 from "int 13h" to "int BBh".   Next, I would create a small .COM loader that would execute the main program as a child process (read the DOS TECH REF on the EXEC function).   But before I did that, I would write an ISR (interrupt service routine) for INT BBh. Here is the general outline for the ISR

þ Use dos to open the file containing the needed data
þ Read in the data to ES:BX (where int 13h would put it)
þ Close the file
þ set AX to 0000 and clear CF
þ iret

The loader would go like this :

þ Get current int BBh address (DOS func. 35h)
þ Set int BBh address to ours (DOS func. 25h)
þ use DOS to EXECUTE (Dos func. 4Bh) the program to be cracked
þ Restore the address of BBh

Well, that about all I have to say about cracking a dos file.   I hope this section has been usefull to you.   Next I will show by example the techinques in this section while cracking I.B.M. Drawing Assistance (1.00).

One last thing.   After you have cracked the program, try running it from a hard drive with PW set to trap calls to INT 21h function 1Bh (get fat byte).   If the program make a call to here, get the address and find that section of code.   What the program is trying to do is check to see if you are running from a hard drive (most programs have diffrent protection routines for hard drives).   When you find it, simply replace the "INT 21h" with a "MOV DS:[BX],FDh".   This will fake the program in to thinking you are working on a floppy disk.

Ok, for our example we will be removing the code from IBM's Drawing assistant.   Now now, I know it's not the best program out there, but shit, It's hard to find shit with on disk copy protection anymore.   So here we go...

I needed 3 programs in cracking the assist. series. Locksmith by Alph Logic, Periscope debugger, and DEBUG (supplied with DOS).   By using these three programs together, I was able to figure out and remove the copy protection in under 30 minutes.

Drawing Assistant (DA) is IBM's answer to colorpaint for the Jr.   It is a simple drawing program (a more advanced version is included in StoryBoard Deluxe) but easy to learn and use.   So far, I have only seen version 1.00 of this program.

DA made calls to the copy protection routine in 3 diffrent modules. The files "SETDRAW.EXE", "DRAWASST.EXE" and "DRAWASST.TWO" all contained calls to the copy protection. Also, "DRAWASST.TWO" and "DRAWASST.EXE" are for all intensive puporses then same file.

I first started off by loading DRAWASST.EXE with debug and searched for any int 13's by executing the debug command

```
        s CS:0 FFFF CD 13             Search CS:0 - CS:FFFF for CD
                                   13 (int 13)
```

I located 2 diffrent calls to int 13h, so I then listed them. Here is what I found...

```
              { First, some initialization routines }

              18FD:0343 1E              PUSH    DS
              18FD:0344 B80000           MOV     AX,0000
              18FD:0347 50              PUSH    AX
              18FD:0348 B89724           MOV     AX,2497
              18FD:034B 8ED8             MOV     DS,AX
              18FD:034D BB1000           MOV     BX,0010
              18FD:0350 2E              CS:
              18FD:0351 8A07             MOV     AL,[BX]
              18FD:0353 3C00             CMP     AL,00
              18FD:0355 7418            JZ      036F


         { This set is called if DA has been installed }
                     { on the hard drive }
           { When cracked, this will NEVER be called }

              18FD:0357 B419             MOV     AH,19
              18FD:0359 CD21             INT     21
              18FD:035B 8AD0             MOV     DL,AL
              18FD:035D FEC2             INC     DL
              18FD:035F B41C             MOV     AH,1C
              18FD:0361 CD21             INT     21
              18FD:0363 8A07             MOV     AL,[BX]
              18FD:0365 BB9724           MOV     BX,2497
              18FD:0368 8EDB             MOV     DS,BX
              18FD:036A 3CF8             CMP     AL,F8
              18FD:036C 7475             JZ      03E3
              18FD:036E CB               RETF

     { This set is called if DA is running from the floppy }

              18FD:036F B419             MOV     AH,19
              18FD:0371 CD21             INT     21
              18FD:0373 FEC0             INC     AL
              18FD:0375 B400             MOV     AH,00
              18FD:0377 A320C6           MOV     [C620],AX
              18FD:037A 8AD0             MOV     DL,AL
              18FD:037C B41C             MOV     AH,1C
```

```
        18FD:037E CD21          INT     21
        18FD:0380 8A07          MOV     AL,[BX]
        18FD:0382 BB9724        MOV     BX,2497
        18FD:0385 8EDB          MOV     DS,BX
        18FD:0387 3CF8          CMP     AL,F8
        18FD:0389 7408          JZ      0393
```

{ Here is the called to read in the key disk }

```
        18FD:038B E8A675        CALL    7934
        18FD:038E 3D0100        CMP     AX,0001
        18FD:0391 7450          JZ      03E3
```

Let's take these code segments 1 at a time. The fist, is some simple initialization routines. Here is the code, only this time full comments are added.

{ First, some initialization routines }
; Setup for return to DOS

```
18FD:0343 1E            PUSH    DS
18FD:0344 B80000        MOV     AX,0000
18FD:0347 50            PUSH    AX
```

; Setup DS to point to the data segment

```
18FD:0348 B89724        MOV     AX,2497
18FD:034B 8ED8          MOV     DS,AX


18FD:034D BB1000        MOV     BX,0010     ; CS:10 points to
                                            ; installed flag
18FD:0350 2E            CS:
18FD:0351 8A07          MOV     AL,[BX]

18FD:0353 3C00          CMP     AL,00       ; If not installed,
                                            ; jump to diskette
18FD:0355 7418          JZ      036F        ; routines
```

What we are want to do, is fool DA in to thinking that it is stilling loading from diskette. Nothing really needs to be changed in this segment, but, just to be safe, we will force the jump at 355. To change the current values, use DEBUG's [A]ssembler command.

```
    A CS:355
    18FD:355 JMP 36F
```

Now, we have forced the jump, we can move on to the third code segment skipping over the second since it will never be called again. The 3rd code segment checks to see if you are using a hard drive. It does so by first getting the logical drive letter, then reading in the FAT descriptor byte for that drive. Here is the commented code.

{ This set is called if DA is running from the floppy }

```
; First, get the current drive

18FD:036F B419           MOV     AH,19       ; DOS function 19h -
                                             ; Get Current Drive
18FD:0371 CD21           INT     21

18FD:0373 FEC0           INC     AL          ; Add 1 for BIOS
18FD:0375 B400           MOV     AH,00       ; Clear AH
18FD:0377 A320C6         MOV     [C620],AX   ; Store it at C620
18FD:037A 8AD0           MOV     DL,AL       ; Store it in DL
18FD:037C B41C           MOV     AH,1C       ; DOS function 1Ch
                                             ; Get Fat desc.
18FD:037E CD21           INT     21          ; returns pointer
                                             ; in DS:BX
18FD:0380 8A07           MOV     AL,[BX]     ; Get the actual
                                             ; byte
18FD:0382 BB9724         MOV     BX,2497     ; Restore DS
18FD:0385 8EDB           MOV     DS,BX

18FD:0387 3CF8           CMP     AL,F8       ; Check to see if
                                             ; it is a H/D
18FD:0389 7408           JZ      0393        ; Yes, then jump
abort

{ Fall in to the check for the key disk }
```

As you can see, this section of code is quite straigth forward. It just checks to see if you are using a hard drive. What we want to do is to fake an DOS function 1Ch and return the value for a floppy. This is done by putting the value of FDh in AL then NOPing the rest. Again use the Debug's [A] command.

```
        A CS:37C

        18FD:037C MOV AL,FD
        18FD:0380 NOP
        18FD:0381 NOP
        18FD:0382 NOP
        18FD:0383 NOP
```

Now, you might ask why I didn't simple force a jump over the code. The answer is what if DA uses the value at C620 at a later time (which it doesn't but let's pretend). If I had forced the jump then the value might not have been initialized and the crack might not work. Now that we have simulated running from diskette, we must deal for the check for the key disk.

This is where Periscope came in to play. Using periscope, I made the above corrections and ran the program up till CS:038B (the call to the check). Here is the code, including the actual check. I have indented the check to make it easier to read.

{ Here is the called to read in the key disk }

```
18FD:038B E8A675            CALL     7934      ; Check key on disk
                                                ; (track 27h side 0)

   18FD:7934 A120C6            MOV     AX,[C620]       ; Get drive
                                                      ; letter
   18FD:7937 FEC8              DEC     AL
   18FD:7939 A23BC6            MOV     [C63B],AL       ; Store it for
                                                      ; later
   18FD:793C 1E                PUSH   DS
   18FD:793D 07                POP     ES
   ; Setup pointers to what sectors to try to read

   18FD:793E BB30C6            MOV     BX,C630
   18FD:7941 891E39C6          MOV     [C639],BX
   18FD:7945 C6063CC603        MOV     BYTE PTR [C63C],03
   18FD:794A C6063DC601        MOV     BYTE PTR [C63D],01

   ; Reset the disk

   18FD:794F B400              MOV     AH,00
   18FD:7951 CD13              INT     13

   ; Get address of sector to read an put it in CL

   18FD:7953 8B1E39C6          MOV     BX,[C639]
   18FD:7957 8A0F              MOV     CL,[BX]


   ; Setup the rest of the read information

   18FD:7959 BBAE3D            MOV     BX,3DAE
   18FD:795C 81C3D007          ADD     BX,07D0
   18FD:7960 B001              MOV     AL,01
   18FD:7962 B527              MOV     CH,27
   18FD:7964 8A163BC6          MOV     DL,[C63B]
   18FD:7968 B600              MOV     DH,00
   18FD:796A B402              MOV     AH,02
   18FD:796C CD13              INT     13

   ; Test for an error and jump if none is present (ie: the
   ; copy protection has been removed)

   18FD:796E 80FC00            CMP     AH,00
   18FD:7971 740C              JZ      797F

   ; test the bad read (protection is missing) 3 times

   18FD:7973 FE0E3CC6          DEC     BYTE PTR [C63C]
   18FD:7977 75D6              JNZ     794F
   18FD:7979 B80000            MOV     AX,0000
   18FD:797C EB13              JMP     7991
   ; Get next sector to check.   If finished, set the flag and
   ; return.

   18FD:797F FF0639C6         INC     WORD PTR [C639]
   18FD:7983 FE063DC6         INC     BYTE PTR [C63D]
   18FD:7987 803E3DC603       CMP     BYTE PTR [C63D],03
   18FD:798C 75C1             JNZ     794F
   18FD:798E B80100           MOV     AX,0001
   18FD:7991 C3               RET

; Check to see if the OK flag was set (ax = 0001h means check
; was good)
18FD:038E 3D0100           CMP     AX,0001
18FD:0391 7450             JZ      03E3
```

The key check used in DA is quite simple. It simple tries to read in the illegaly numbered sectors on Track 27h side 0h. If they are missing, it assumes that it is running a pirated copy. What we must do, is to fool the scheme in to thinking a good read happened. I choses to fake the read using the easiest method. Since the protection scheme only check to see if AX returns the value > 0000h, I simply modified the routine at 1BFD:7934 to set AX to 0000h and then return. Here is the new code (enter using debug's A command)...

```
        A 1BFD:7934
        1BFD:7934 MOV AX,0000
        1BFD:7936 RET
```

Now, this file is unprotected and if you type "G" at debug's command prompt, the program will execute, sort-of. See DRAWASST.EXE calls DRAWASST.TWO which also has the protection scheme. So both must be changed. To make to changes perement in DRAWASST.EXE, rename the file to DRAWASST.DEB and edit it. To find the address of the start of the protection code, use debug's search command...

```
        S CS:0 FFFF B4 19 CD 21 8A D0
```

Now, just uses the modified address to change the program (the code will still be the same, just all calls and jumps will be to diffrent addresses). Use the same process to stip DRAWASST.TWO (it uses the exact same code). When you have both of those files unprotected, you can move on to unprotecting the setup program "SETDRAW.EXE"

DRAWASST.EXE AND .TWO are not the only programs that make calls to the protection routine. SETDRAW.EXE also makes the above calls. Although the check here is much easier to bypass. Here is the asm listing of SETDRAW with all of the calls to the protection. This time, I will not go in to quite as much detail as I did for the other two version.

I will tell you this. When SETDRAW checks the key disk, first it checks to see if the protection exists and then to see if the track hasn't been modified. It again uses AX to determine what happeded. I used Periscope to trace through the original version to find out what the correct values are.
Here is the asm code...

```
; Initialization - checks the current mode and saves it.

18FD:0000 1E              PUSH    DS
18FD:0001 B80000          MOV     AX,0000
18FD:0004 50              PUSH    AX
18FD:0005 B8321A          MOV     AX,1A32
18FD:0008 8ED8            MOV     DS,AX
18FD:000A B40F            MOV     AH,0F
18FD:000C CD10            INT     10
18FD:000E 3C02            CMP     AL,02
18FD:0010 740D            JZ      001F
18FD:0012 3C03            CMP     AL,03
18FD:0014 7409            JZ      001F
18FD:0016 A28900          MOV     [0089],AL
18FD:0019 B002            MOV     AL,02
18FD:001B B400            MOV     AH,00
18FD:001D CD10            INT     10

; Gets the current drive

18FD:001F B400            MOV     AH,00
18FD:0021 B419            MOV     AH,19
18FD:0023 CD21            INT     21
18FD:0025 A28700          MOV     [0087],AL
18FD:0028 8AD0            MOV     DL,AL
18FD:002A FEC2            INC     DL

; Checks the FAT descriptor

18FD:002C B41C            MOV     AH,1C
18FD:002E CD21            INT     21
18FD:0030 8A07            MOV     AL,[BX]
18FD:0032 BB321A          MOV     BX,1A32
18FD:0035 8EDB            MOV     DS,BX
18FD:0037 C606880000      MOV     BYTE PTR [0088],00
18FD:003C 3CF8            CMP     AL,F8
18FD:003E 742A            JZ      006A

; Read in protection scheme

18FD:0040 8A168700        MOV     DL,[0087]
18FD:0044 E87E0A          CALL    0AC5
18FD:0047 C606880000      MOV     BYTE PTR [0088],00
18FD:004C 3D0000          CMP     AX,0000
18FD:004F 7419            JZ      006A

; Read in the dummy scheme

18FD:0051 C606880001      MOV     BYTE PTR [0088],01
```

```
18FD:0056 8A168700         MOV         DL,[0087]
18FD:005A B84500           MOV         AX,0045
18FD:005D E8BD0A           CALL        0B1D
18FD:0060 3D0000           CMP         AX,0000
18FD:0063 7405             JZ          006A

; Start of actual routine.

18FD:0065 C606880000       MOV         BYTE PTR [0088],00
```

There is isn't much to say about the above code. To bypass it, we will change the hard drive check (int 21 function 1c). Do the same thing we did for DRAWASST.EXE

```
       A 18FD:2C
       18FD:002C mov AL,FD
       18FD:002E nop
       18FD:002F nop
       18FD:0030 nop
       18FD:0031 nop
```

Now, just jump over the check to the key disk.

```
      A 18FD:40

      18FD:0040 jmp 0065
```

And thats it. Now SETDRAW is unprotected. Drawing Assistant may be used, copied or backed up at your pleasure.

As you can see, this was a good example although the fact that if you only made the changes in DRAWASST.EXE and not in DRAWASST.TWO then the program would copy DRAWASST.TWO to DRAWASST.EXE to restore the protection was a bit strange.

Well, I hope you are proud. But be warned, next we take on DOC checks, so get a good nights sleep. Till then, play lots of SMASH T.V.

-Buckaroo Banzai

CRACKING 101 - 1990 edition

Lesson 2

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ DOC CHECK PRIMER ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

by Buckaroo Banzai

Ok, in this textfile, I will start talking about removing doc check protection schemes. I find, the doc check scheme to be slightly more difficult to work on than normal INT 13 schemes.

What is a doc check. Usually, a doc check when a program ask the user to enter a phrase or code supplied with the manual. Now, one might think that "Shit, we can just type all the codes in to a textfile and upload it with the DOCS", but that way of thinking breaks down on programs such as Future Classics where there are 6 pages with about 200 codes per page. So it is just better to remove the check completely.

In this primer, I will get in to the theory of removing a doc check, then start with a simple example (Electronic Art's ESCAPE FROM HELL). Then in the next file, I will take you deeper in to the world of doc checks and work with more difficult examples. But for now, lets get started.

A doc check, in basic theory works much like normal INT 13 copy protection. Somewhere in the beginning of the program before it really starts, the check is made. If the result is ok (ie the user enters the correct word or phrase) then the program continues. If not, then the program simply exits to dos.

Simple right, well not really. Usually, the input routine is part of the standard input routine of the program so you just can't go about modify the call to INT 16h (the keyboard interrupt) like you could with INT 13h. So, where do we start. If you think back to cracking the old INT 13 protection schemes, you would use a program like PCWATCH or TRAP13 to get a rough idea of where the call resides. With doc checks, this is really not the best way to do it.

I suggest that you try to break in to the program well before the protection is checked. Remember, we must remove the check without messing with the actual input routine so we want to come in highest level.

So, how do we break in. By using a good debugger. I suggest Periscope. I find it is the best and easiest to use. Once we are in, all the is left is to trace through the program until we find the topmost call to the doc check. Now we're moving.

So let's say we have broken in to the program and found the topmost call to the doc check. What next. We must try to figure out what the program does. There are 2 possibilities. First, the program could simply check the inputed string against a value in memory, and if they don't match simply exit to dos and if they do, just continue with the program; or if the input matches it can set a flag in memory that is checked by some routine later.

So, on to the example. NOTE! All address might be different. This is how it looked when I cracked it. ALSO NOTE, you should be cracking without any memory resident programs. Make sure MEMORY is clear, and that you load the system the same way each time. Remember, if you load everything the same, everything will be in the same memory location.

So, what is our first step. Well, I suggest picking out the right tools to do the job. In this case, You will only need PERISCOPE (and the addin program that comes with it called PSKEY) and a good file editor (when I say good I mean it can edit and search in hex). So let's get started.

First, we load PERISCOPE (PS from now on). This is gonna be the debugger we use. Next, we need a quick way in to the debugger. Since ESCAPE FROM HELL (EFH from now on) is not all the picky about how it keeps a crackest out, PSKEY will do just fine but not without using a little trick.

Normally, when using PSKEY (for those of you who do not know what PSKEY does, it allows up to break in to PS usings a TSR hotkey) and you hit the hotkey, PSKEY does an INT 2h (NMI). This then brings up PS and you are set. But, EFH revectors INT 2h (NMI) to simply an IRET so this method does not work. How do we get around this, well, INT 2h is the default used with PSKEY but not the only way to work it. You can also use INT 3h (Breakpoint interrupt) or INT 15h (Extended services interrrupt) to activate PS. In this case we will use INT 3h; so when we invoke PSKEY we add the command line parameter "3" (ie: "PSKEY 3CAL" invokes PSKEY using INT 3h setting the hotkey to CTRL-ALT-LEFT_SHIFT).

So, now that we have a way in to EFH, where do we want to break out. Well boys (and Girls, and BTW: if there are any Fems reading this, give me a ring, I'd like to hear from ya) I don't have any formula to give, but remember, I suggest that we try to break in to the outermost loop. So, experiance (and a good fucking guess) tells me to break out in the title screen before the music begins.

It just so happens that this time I was right (And noone had to get nail to anything -D.A.) Right after the title picture comes up, press your hotkey (oooh). The PS debugging screen should come up and you should see the follow section of code..

```
2309:019C CF            IRET
2309:019D 3D0085         CMP AX,8500
etc.
```

This is the exit code from PSKEY. By usings the J(ump) command, and executing the IRET, you will be put back right to the spot where you pressed the hotkey (boy I'm getting excited). I would love to give you a code fragment here, but each time you press the hotkey you will end up at a different point.

So what do we do next. Well, we will just have to keep executing code until we find some reference point. Remember how I said we wanted to break out before we reached to music at the title screen. Well, you can bet that we are in the outermost loop since the music comes before the doc check and we haven't reached the call to the music routine yet. So we start executing code.

Then all of the sudden BOOM! you execute a CALL instruction and music bursts through the speaker. AHa, a reference point. We know we are on the right track.

Press <ESC> during the music so that we can skip the stupid intro for now. After pressing <ESC> you should regain control at the instruction after the call to the music routine.

From here on out, we want to procede rather slowly. Each time you reach a CALL instruction you want to write down the address where it is located. Sooner or later you will execute a CALL instruction and EFH will jump in to it's doc check routine. But damn, you have the address of the that call WRITTEN DOWN right. So simply reboot and reload everything.

Break out in to PS at the title picture. Now, unassemble the address you wrote down. You should see something like this

```
21DD:3EA4 9AA5368132      CALL 3281:36A5   (current line)
21DD:3EA4 9A522F8132      CALL 3281:2F52
21DD:3EA4 C706BB070000    MOV WORD PTR [07BB],0000
21DD:3EA4 8BE5            MOV SP,BP
21DD:3EA4 5D              POP BP
21DD:3EA4 CB              RETF
```

The first call, is the call to the doc check, therefore it can for now be assumed that the second call is to the actual game (remember, most programmers follow good programming practice and will exit the routine that does the doc check to finish the game). Please NOTE, from here on out, if I say go back to STEP 1, reboot the machine, reload and get to this point. Ok.

Our first though in seeing code like this is shit maybe they just check the keyword and exit to dos if it's bad; if it it's good, then they just exit that subroutine and start the game. So having lots of time on our hands, we try just executing the second CALL and bypass the first (you can do that by setting the IP (instruction pointer) register to the offset of the second call [In our case 3EA9]). When you do this, the screen clears, and you see the character (Richard) on the screen. But just as you think it worked, it switches back to text mode and prints the message "Hell is HOT". Shit I hate it when that happens.

So now we know that somewhere in the doc check routine, EFH sets a flag in memory. We must figure out where this flag is and figure out a way to fake it. So go back to step 1, this time, let's trace (using the T command) in to the doc check routine.

I have included the entire outerloop of the doc check routine here. The small subroutines are not of any importants and infact when I first crack EFH, I never traced in to any of them. It wasn't until I was out getting this information that I took a look to see what they did.

Here is the dos check code. I have place some basic instructions that should help you as you go along. Although you address might be different than mine, I will use mine for reference. Also, I have noted some special subroutines along the way.

(- Unassembled DOC CHECK for ESCAPE FROM HELL [outer loop])

First, we start off with some initialization routines. You don't need to be all to concerned with them.

```
3281:36A5 55              PUSH BP
3281:36A6 8BEC            MOV BP,SP
3281:36A8 83EC2A          SUB SP,+2A
3281:36AB C746DE0000      MOV WORD PTR [BP-22],0000
3281:36B0 B80600          MOV AX,0006
3281:36B3 50              PUSH AX
3281:36B4 9AE3169900      CALL 0099:16E3
3281:36B9 59              POP CX
3281:36BA 48              DEC AX
3281:36BB 8946DA          MOV [BP-26],AX
3281:36BE B80F00          MOV AX,000F
3281:36C1 50              PUSH AX
3281:36C2 9AE3169900      CALL 0099:16E3
3281:36C7 59              POP CX
3281:36C8 48              DEC AX
3281:36C9 8946DC          MOV [BP-24],AX
3281:36CC C706CB070E00    MOV WORD PTR [07CB],000E
3281:36D2 C746D60000      MOV WORD PTR [BP-2A],0000
3281:36D7 E9C002          JMP 399A
3281:36DA C746D80000      MOV WORD PTR [BP-28],0000
3281:36DF E92501          JMP 3807
3281:36E2 9A9B479900      CALL 0099:479B
3281:36E7 B83866          MOV AX,6638
3281:36EA 50              PUSH AX
3281:36EB A03407          MOV AL,[0734]
3281:36EE B400            MOV AH,00
3281:36F0 50              PUSH AX
3281:36F1 B80C00          MOV AX,000C
3281:36F4 50              PUSH AX
3281:36F5 B8CF00          MOV AX,00CF
```

```
3281:36F8 50              PUSH AX
3281:36F9 8B46DC          MOV AX,[BP-24]
3281:36FC BA5800          MOV DX,0058
3281:36FF F7E2            MUL DX
3281:3701 8BD8            MOV BX,AX
3281:3703 8A87F640        MOV AL,[BX+40F6]
3281:3707 B400            MOV AH,00
3281:3709 8BD8            MOV BX,AX
3281:370B 81C39400        ADD BX,0094
3281:370F D1E3            SHL BX,1
3281:3711 D1E3            SHL BX,1
3281:3713 FFB7F25D        PUSH [BX+5DF2]
3281:3717 FFB7F05D        PUSH [BX+5DF0]
3281:371B 9AE7019900      CALL 0099:01E7
3281:3720 83C40C          ADD SP,+0C
3281:3723 8B46DA          MOV AX,[BP-26]
3281:3726 3D0500          CMP AX,0005
3281:3729 7603            JBE 372E
3281:372B E9B200          JMP 37E0
3281:372E 8BD8            MOV BX,AX
3281:3730 D1E3            SHL BX,1
3281:3732 2E              CS:
3281:3733 FFA73737        JMP [BX+3737]
3281:3737 43              INC BX
3281:3738 37              AAA
3281:3739 5E              POP SI
3281:373A 37              AAA
3281:373B 7837            JS 3774
3281:373D 92              XCHG DX,AX
3281:373E 37              AAA
3281:373F AC              LODSB
3281:3740 37              AAA
3281:3741 C637B8          MOV BYTE PTR [BX],B8
3281:3744 2000            AND [BX+SI],AL
3281:3746 50              PUSH AX
3281:3747 B82E01          MOV AX,012E
3281:374A 50              PUSH AX
3281:374B B88100          MOV AX,0081
3281:374E 50              PUSH AX
3281:374F B87348          MOV AX,4873
3281:3752 50              PUSH AX
3281:3753 9AD6029900      CALL 0099:02D6
3281:3758 83C408          ADD SP,+08
3281:375B E98200          JMP 37E0
3281:375E B82000          MOV AX,0020
3281:3761 50              PUSH AX
3281:3762 B82E01          MOV AX,012E
3281:3765 50              PUSH AX
3281:3766 B88100          MOV AX,0081
3281:3769 50              PUSH AX
3281:376A B88648          MOV AX,4886
3281:376D 50              PUSH AX
3281:376E 9AD6029900      CALL 0099:02D6
3281:3773 83C408          ADD SP,+08
3281:3776 EB68            JMP 37E0
3281:3778 B82000          MOV AX,0020
```

```
3281:377B 50            PUSH AX
3281:377C B82E01        MOV AX,012E
3281:377F 50            PUSH AX
3281:3780 B88100        MOV AX,0081
3281:3783 50            PUSH AX
3281:3784 B8AD48        MOV AX,48AD
3281:3787 50            PUSH AX
3281:3788 9AD6029900    CALL 0099:02D6
3281:378D 83C408        ADD SP,+08
3281:3790 EB4E          JMP 37E0
3281:3792 B82000        MOV AX,0020
3281:3795 50            PUSH AX
3281:3796 B82E01        MOV AX,012E
3281:3799 50            PUSH AX
3281:379A B88100        MOV AX,0081
3281:379D 50            PUSH AX
3281:379E B8C748        MOV AX,48C7
3281:37A1 50            PUSH AX
3281:37A2 9AD6029900    CALL 0099:02D6
3281:37A7 83C408        ADD SP,+08
3281:37AA EB34          JMP 37E0
3281:37AC B82000        MOV AX,0020
3281:37AF 50            PUSH AX
3281:37B0 B82E01        MOV AX,012E
3281:37B3 50            PUSH AX
3281:37B4 B88100        MOV AX,0081
3281:37B7 50            PUSH AX
3281:37B8 B8E848        MOV AX,48E8
3281:37BB 50            PUSH AX
3281:37BC 9AD6029900    CALL 0099:02D6
3281:37C1 83C408        ADD SP,+08
3281:37C4 EB1A          JMP 37E0
3281:37C6 B82000        MOV AX,0020
3281:37C9 50            PUSH AX
3281:37CA B82E01        MOV AX,012E
3281:37CD 50            PUSH AX
3281:37CE B88100        MOV AX,0081
3281:37D1 50            PUSH AX
3281:37D2 B80F49        MOV AX,490F
3281:37D5 50            PUSH AX
3281:37D6 9AD6029900    CALL 0099:02D6
3281:37DB 83C408        ADD SP,+08
3281:37DE EB00          JMP 37E0
3281:37E0 B82D00        MOV AX,002D
3281:37E3 50            PUSH AX
3281:37E4 B88200        MOV AX,0082
3281:37E7 50            PUSH AX
3281:37E8 9A96029900    CALL 0099:0296
3281:37ED 59            POP CX
3281:37EE 59            POP CX
3281:37EF B82849        MOV AX,4928
3281:37F2 50            PUSH AX
3281:37F3 9A3F039900    CALL 0099:033F
3281:37F8 59            POP CX
3281:37F9 837ED800      CMP WORD PTR [BP-28],+00
3281:37FD 7505          JNZ 3804
```

Here is the first point of interest.   The call on the following line will display the "what is xxxx" message. Ä¿
                                                                 ³
```
3281:37FF 9A1B019900      CALL 0099:011B <ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ

3281:3804 FF46D8           INC WORD PTR [BP-28]
3281:3807 837ED802         CMP WORD PTR [BP-28],+02
3281:380B 7D03             JGE 3810
3281:380D E9D2FE           JMP 36E2
3281:3810 8B46DA           MOV AX,[BP-26]
3281:3813 3D0500           CMP AX,0005
3281:3816 7603             JBE 381B
3281:3818 E97401           JMP 398F
3281:381B 8BD8             MOV BX,AX
3281:381D D1E3             SHL BX,1
3281:381F 2E               CS:
3281:3820 FFA72438         JMP [BX+3824]
3281:3824 3038             XOR [BX+SI],BH
3281:3826 6E               DB 6E
3281:3827 38AC38EA         CMP [SI+EA38],CH
3281:382B 3827             CMP [BX],AH
3281:382D 396439           CMP [SI+39],SP
3281:3830 B81000           MOV AX,0010
3281:3833 50               PUSH AX
3281:3834 16               PUSH SS
3281:3835 8D46E2           LEA AX,[BP-1E]
3281:3838 50               PUSH AX
3281:3839 9AFB149900       CALL 0099:14FB
3281:383E 83C406           ADD SP,+06
3281:3841 8D46E2           LEA AX,[BP-1E]
3281:3844 50               PUSH AX
3281:3845 9A0F00B81B       CALL 1BB8:000F
3281:384A 59               POP CX
3281:384B 8B46DC           MOV AX,[BP-24]
3281:384E BA5800           MOV DX,0058
3281:3851 F7E2             MUL DX
3281:3853 05F740           ADD AX,40F7
3281:3856 50               PUSH AX
3281:3857 8D46E2           LEA AX,[BP-1E]
3281:385A 50               PUSH AX
3281:385B 9A0E00661A       CALL 1A66:000E
3281:3860 59               POP CX
3281:3861 59               POP CX
3281:3862 0BC0             OR AX,AX
3281:3864 7505             JNZ 386B
3281:3866 C746DEFFFF       MOV WORD PTR [BP-22],FFFF
3281:386B E92101           JMP 398F
3281:386E B81000           MOV AX,0010
3281:3871 50               PUSH AX
3281:3872 16               PUSH SS
3281:3873 8D46E2           LEA AX,[BP-1E]
3281:3876 50               PUSH AX
3281:3877 9AFB149900       CALL 0099:14FB
3281:387C 83C406           ADD SP,+06
3281:387F 8D46E2           LEA AX,[BP-1E]
```

```
3281:3882 50              PUSH AX
3281:3883 9A0F00B81B      CALL 1BB8:000F
3281:3888 59              POP CX
3281:3889 8B46DC          MOV AX,[BP-24]
3281:388C BA5800          MOV DX,0058
3281:388F F7E2            MUL DX
3281:3891 050841          ADD AX,4108
3281:3894 50              PUSH AX
3281:3895 8D46E2          LEA AX,[BP-1E]
3281:3898 50              PUSH AX
3281:3899 9A0E00661A      CALL 1A66:000E
3281:389E 59              POP CX
3281:389F 59              POP CX
3281:38A0 0BC0            OR AX,AX
3281:38A2 7505            JNZ 38A9
3281:38A4 C746DEFFFF      MOV WORD PTR [BP-22],FFFF
3281:38A9 E9E300          JMP 398F
3281:38AC B81000          MOV AX,0010
3281:38AF 50              PUSH AX
3281:38B0 16              PUSH SS
3281:38B1 8D46E2          LEA AX,[BP-1E]
3281:38B4 50              PUSH AX
3281:38B5 9AFB149900      CALL 0099:14FB
3281:38BA 83C406          ADD SP,+06
3281:38BD 8D46E2          LEA AX,[BP-1E]
3281:38C0 50              PUSH AX
3281:38C1 9A0F00B81B      CALL 1BB8:000F
3281:38C6 59              POP CX
3281:38C7 8B46DC          MOV AX,[BP-24]
3281:38CA BA5800          MOV DX,0058
3281:38CD F7E2            MUL DX
3281:38CF 051941          ADD AX,4119
3281:38D2 50              PUSH AX
3281:38D3 8D46E2          LEA AX,[BP-1E]
3281:38D6 50              PUSH AX
3281:38D7 9A0E00661A      CALL 1A66:000E
3281:38DC 59              POP CX
3281:38DD 59              POP CX
3281:38DE 0BC0            OR AX,AX
3281:38E0 7505            JNZ 38E7
3281:38E2 C746DEFFFF      MOV WORD PTR [BP-22],FFFF
3281:38E7 E9A500          JMP 398F
3281:38EA B81000          MOV AX,0010
3281:38ED 50              PUSH AX
3281:38EE 16              PUSH SS
3281:38EF 8D46E2          LEA AX,[BP-1E]
3281:38F2 50              PUSH AX
3281:38F3 9AFB149900      CALL 0099:14FB
3281:38F8 83C406          ADD SP,+06
3281:38FB 8D46E2          LEA AX,[BP-1E]
3281:38FE 50              PUSH AX
3281:38FF 9A0F00B81B      CALL 1BB8:000F
3281:3904 59              POP CX
3281:3905 8B46DC          MOV AX,[BP-24]
3281:3908 BA5800          MOV DX,0058
3281:390B F7E2            MUL DX
```

```
3281:390D 052A41           ADD AX,412A
3281:3910 50               PUSH AX
3281:3911 8D46E2           LEA AX,[BP-1E]
3281:3914 50               PUSH AX
3281:3915 9A0E00661A       CALL 1A66:000E
3281:391A 59               POP CX
3281:391B 59               POP CX
3281:391C 0BC0             OR AX,AX
3281:391E 7505             JNZ 3925
3281:3920 C746DEFFFF       MOV WORD PTR [BP-22],FFFF
3281:3925 EB68             JMP 398F
3281:3927 B81000           MOV AX,0010
3281:392A 50               PUSH AX
3281:392B 16               PUSH SS
3281:392C 8D46E2           LEA AX,[BP-1E]
3281:392F 50               PUSH AX
```

Next point of interest. When you execute this line, the game will pause and wait for you to enter the code word from the manual. ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿

```
                                                 ³
                                                 ³
3281:3930 9AFB149900       CALL 0099:14FB <ÄÄÄÄÄÙ

3281:3935 83C406           ADD SP,+06
3281:3938 8D46E2           LEA AX,[BP-1E]
3281:393B 50               PUSH AX
3281:393C 9A0F00B81B       CALL 1BB8:000F
3281:3941 59               POP CX
3281:3942 8B46DC           MOV AX,[BP-24]
3281:3945 BA5800           MOV DX,0058
3281:3948 F7E2             MUL DX
3281:394A 053B41           ADD AX,413B
3281:394D 50               PUSH AX
3281:394E 8D46E2           LEA AX,[BP-1E]
3281:3951 50               PUSH AX
3281:3952 9A0E00661A       CALL 1A66:000E
3281:3957 59               POP CX
3281:3958 59               POP CX
3281:3959 0BC0             OR AX,AX
3281:395B 7505             JNZ 3962
3281:395D C746DEFFFF       MOV WORD PTR [BP-22],FFFF
3281:3962 EB2B             JMP 398F
3281:3964 33D2             XOR DX,DX
3281:3966 B8B80B           MOV AX,0BB8
3281:3969 52               PUSH DX
3281:396A 50               PUSH AX
```

Next point of interest. This call is the final evaluation of the entered word (or phrase). On return, it checks a checksum value. This whole next section of code (up to 3281:39Ad) simply test the validity of the keyword you entered. I have marked the all jumps that happened when I entered my keyword with an " * ".

```
3281:396B 9A71139900       CALL 0099:1371
```

```
3281:3970 59                 POP CX
3281:3971 59                 POP CX
3281:3972 8946E0             MOV [BP-20],AX
3281:3975 8B46DC             MOV AX,[BP-24]
3281:3978 BA5800             MOV DX,0058
3281:397B F7E2               MUL DX
3281:397D 8BD8               MOV BX,AX
3281:397F 8B874C41           MOV AX,[BX+414C]
3281:3983 3B46E0             CMP AX,[BP-20]
3281:3986 7505              *JNZ 398D
3281:3988 C746DEFFFF         MOV WORD PTR [BP-22],FFFF
3281:398D EB00               JMP 398F
3281:398F 837EDE00           CMP WORD PTR [BP-22],+00
3281:3993 7402              *JZ 3997
3281:3995 EB0C               JMP 39A3
3281:3997 FF46D6             INC WORD PTR [BP-2A]
3281:399A 837ED602           CMP WORD PTR [BP-2A],+02
3281:399E 7D03              *JGE 39A3
3281:39A0 E937FD             JMP 36DA
3281:39A3 837EDE00           CMP WORD PTR [BP-22],+00
3281:39A7 7504              *JNZ 39AD
3281:39A9 0E                 PUSH CS
3281:39AA E8E8FC             CALL 3695
```

This is the last point of interest. The next instruction is where we set the key (by moving FFFFh to the memory location DS:0744h). This is what we need to fake to allow the system to run. ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿

```
                                                    ³
3281:39AD C7064407FFFF   MOV WORD PTR [0744],FFFF <ÄÄÄÙ
3281:39B3 B8FFFF             MOV AX,FFFF
3281:39B6 50                 PUSH AX
3281:39B7 9AC0479900         CALL 0099:47C0
3281:39BC 59                 POP CX
3281:39BD 8BE5               MOV SP,BP
3281:39BF 5D                 POP BP
3281:39C0 CB                 RETF
```

Ok, we have now finished the doc check, and control has returned (when the RETF instruction was executed) to 21DD:3EA9. We are now ready to continue with the game.

Notice the instruction at 3281:39AD. This is where EFH sets that external flag. But how did I determine this. Well, by luck. If you look through the entire routine, you will not find any other instructions placing a value in the data segment (DS). And since I decided a long time ago that EFH was written in a higher level language, we can assume that it is writting to some variable.

So, hoping that we have found the flag, we go back to step 1. This time, we manualy edit the word at DS:0744 and place the value FFFFh there. We then skip over the call the the doc check and execute the game. Then before our eyes, shit happenes. The game comes up, and everything is fine.

By George you've got it.

So how do we fix the program to always return a good doc check. Well, we could go about it 2 ways. The first, is you could simple modify the instruction at 3281:3935 to perform a long jump to 3281:39AD. This would force set the value no matter what was typed. But who the fuck wants to have to type anything. I sure don't so lets think of another way.

If we look at the entire doc check routine, we will see that it does nothing but handle the doc check (remember when we first bypassed the check. The screen came up and everything looked fine until it dropped you out. So we can assume that the actual screen is not setup in doc check. So I suggest placing a small patch right in the begining of the doc check.

But what should this patch do? (BTW: it's late and I don't know If I'm using ?s right. So if not TOO FUCKING BAD). Well, all it should do is place the value FFFFh at DS:0744h. Here is the assembly language routine to do it.

```
50         PUSH AX
B8FFFF     MOV AX,FFFF
3E         DS:
A34407     MOV WORD PTR [0744],AX
58         POP AX
CB         RETF
```

This small routine will place the value FFFFh at DS:744 and then exit back to the main loop. Simple huh (note, you don't really need the save AX or load AX with FFFFh for that matter but I did it for clarity).

So now that we have the patch, and now where to put it, how do we get it there. Well, thats where the file editor comes in, but first you will need 2 things. The hex equivlent of out patch (in this case the 10 bytes : 50,B8,FF,FF,3E,A3,44,07,58,CB) and some string to search for. I suggest usings the first 14 bytes of the routines we are going to write over (the code at address 3281:36A5). Those bytes are 55, 8B, EC, 83, EC ,2A ,C7, 46, DE ,00 ,00, B8, 06, and 00. When selecting the search string, select only instructions that ARN'T call, jump, loop or any instruction that has a memory address in them. This value will NOT be the same when you do the search.

Now, using for file editor (I used PCTOOLS, but NORTON's will do) search for our string (55,8B, etc). When it is found (somewhere near sector 200) write down the sector #. Now, go and edit that sector. Find our search string (55,8B, etc) and replace it with the patch string (50,B8,FF, etc). Now save the sector.

Your down. Try playing the game. It should load up, and then go right from the title page (or the intro) to the game without stopping at the doc check. If your doesn't, then you fucked up. Restart from the beginning (NO, this file didn't fuck up, and I DON'T MAKE MISTAKES).

Well, you did it. You have now removed your first doc check. Don't ya feel real good. With time, you will be able to remove any type of doc check.

-BUCKAROO BANZAI

At this time I would just like to say

`ALL CRACKING GROUPS SUCK!'

CRACKING 101 - 1990 edition

Lesson 3

ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ CHAMBER OF THE SCI-MUTANT PREISTEST ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ

Oh shit, I have finally found a newer program that has on disk copy protection. Good, you'all need a refresher course on so here it is (YO JB study hard, you might learn something).

CHAMBER of the SCI-MUTANT PREISTEST (CSMP) is a really fucked up game but was simple to unprotect. So, lets dive right in. We will be using DEBUG here (although I used periscope but then shit I'm special) to do the crack. Lets dive in. When we first load CSMP (the file ERE.COM) and unassemble it here is what we get.

```
u 100 10B

119A:0100 8CCA            MOV DX,CS
119A:0102 81C2C101        ADD DX,01C1
119A:0106 52             PUSH DX
119A:0107 BA0F00          MOV DX,000F
119A:010A 52             PUSH DX
119A:010B CB              RETF
```

I included the register listing for a reason. NOTICE that this piece of code just seem to stop (the RETF) statement. Well, what is really does is place the address (segment and offset) of the real starting point on to the stack and the execute a far return to that location. Now this might fool a real beginner (or at least make him worry a bit but us...no way).

If you take the current CS value and add 1C1 to it (in segment addition) you will get the segment address 135B (that is if you are using my example of 119A. If not then you will not get 135B but trust me, it's the right value).

So since we want to be at the real program, execute the code until 10B (ie use the command "G 10B") then trace through the next instruction.

If you now unassemble the code, here is what it should look like.

```
-u 000f 36

135B:000F 9C            PUSHF
135B:0010 50            PUSH AX
135B:0011 1E            PUSH DS
135B:0012 06            PUSH ES
135B:0013 0E            PUSH CS
135B:0014 1F            POP DS
135B:0015 0E            PUSH CS
135B:0016 07            POP ES
135B:0017 FC            CLD
135B:0018 89260B00      MOV [000B],SP
135B:001C C70600000102  MOV WORD PTR [0000],0201
135B:0022 B013          MOV AL,13
135B:0024 A23500        MOV [0035],AL
135B:0027 A2FF01        MOV [01FF],AL
135B:002A A22F02        MOV [022F],AL
135B:002D A23901        MOV [0139],AL
135B:0030 B280          MOV DL,80
135B:0032 B408          MOV AH,08
135B:0034 CD21          INT 21
135B:0036 7232          JB 006A
```

Since we are looking for a disk based copy protection, it might be a good time to look for INT 13. So search the current segment for INT 13 with the command

```
S 135B:0 FFFF CD 13
```

But shit, nothing. You mean this program doesn't use int 13. Be real. Reread the first lesson. You know the one that talks about self modifing code. This is what we have here. Let's take a closer look at the last bit of code but this time, with my comments added.

```
-u 000f 36

; The first part of the code simple sets up for the return to
; dos as well as sets ES and DS

135B:000F 9C              PUSHF
135B:0010 50              PUSH AX
135B:0011 1E              PUSH DS
135B:0012 06              PUSH ES
135B:0013 0E              PUSH CS
135B:0014 1F              POP DS        ; Set DS to CS
135B:0015 0E              PUSH CS
135B:0016 07              POP ES        ; Set ES to DS
135B:0017 FC              CLD

135B:0018 89260B00        MOV [000B],SP

; The next instruction sets up a variable that is used in the
; routine that reads in the sectors from the disk.   More on
; later.

135B:001C C70600000102   MOV WORD PTR [0000],0201

; Now, here is the self modifing code.   Notice at AL is 13
; (INT 13h ... Get it).   Look at the first memory location
; (35h) and remember that DS = CS.   With this in mind, when
; then instuction at 135B:0024 is executed byte at 135B:0035
; will be changed to 13h.   That will in fact change the
; INT 21h at 135B:0034 to an INT 13h.   And so on, and so on.

135B:0022 B013            MOV AL,13        ; New value
135B:0024 A23500          MOV [0035],AL    ; Change to INT 13h
135B:0027 A2FF01          MOV [01FF],AL    ; Change to INT 13h
135B:002A A22F02          MOV [022F],AL    ; Change to INT 13h
135B:002D A23901          MOV [0139],AL    ; Change to INT 13h

; If you lookup DOS function 08 you will find it's CONSOLE
; INPUT.   Now does that seem out of place to you.

135B:0030 B280            MOV DL,80
135B:0032 B408            MOV AH,08
135B:0034 CD21            INT 21      ; Changed to INT 13h
135B:0036 7232            JB 006A
```

Whoa, that was tricky.   If you execute up to 135B:30 here is what it should look like..

```
135B:0030 B280            MOV DL,80
135B:0032 B408            MOV AH,08
135B:0034 CD13            INT 13
135B:0036 7232            JB 006A
```

AHA, now we are getting somewhere.   If we lookup what disk function 08 means, you won't be suprised.   Function 08h is GET DRIVE TYPE.   It will tell what type of disk drive we have.   Remember, if you are loading off of a hard disk then it wants to use a different routine.   Since we want it to think we are loading off of disk, then we want to take this jump. So for now, force the jmp by setting IP to 6A.

At 135B:006A you find another jmp instruction

```
135B:006A EB6B              JMP 00D7
```

This jumps to the routine that does the actual disk check. Here is the outer loop of that code (With my comments of course).

```
; This first part of this routine simply test to see how many
; disk drives you have.


135B:00D7 CD11              INT 11
135B:00D9 25C000            AND AX,00C0
135B:00DC B106              MOV CL,06
135B:00DE D3E8              SHR AX,CL
135B:00E0 FEC0              INC AL
135B:00E2 FEC0              INC AL
135B:00E4 A20200            MOV [0002],AL

; Next, so setup for the actual disk check


135B:00E7 C606090000        MOV BYTE PTR [0009],00
135B:00EC B9F127            MOV CX,27F1
135B:00EF 8BE9              MOV BP,CX
135B:00F1 B107              MOV CL,07
135B:00F3 F8                CLC

; This calls the protection routine part 1

135B:00F4 E82F00            CALL 0126

135B:00F7 B9DE27            MOV CX,27DE
135B:00FA 8BE9              MOV BP,CX
135B:00FC B108              MOV CL,08
135B:00FE F9                STC

; This calls the protection routine part 2

135B:00FF E82400            CALL 0126

135B:0102 8D1E5802          LEA BX,[0258]
135B:0106 8D361C01          LEA SI,[011C]
135B:010A 8BCD              MOV CX,BP
135B:010C AC                LODSB
135B:010D 8AC8              MOV CL,AL

; This calls the protection routine part 3

135B:010F E8E300            CALL 01F5
```

```
; Makes the final check

135B:0112 7271             JB 0185
135B:0114 AC               LODSB
135B:0115 0AC0             OR AL,AL
135B:0117 75F4             JNZ 010D   ; If not correct, try again
135B:0119 EB77             JMP 0192   ; Correct, continue program
135B:011B 90               NOP
```

There are calls to 2 different subroutines. The routine at 126 and the routine at 1F5. If you examine the routine at 126 you find that it makes several calls to the routine at 1F5. Then you you examine the routine at 1F5 you see the actual call to INT 13. Here is the code for both routine with comments

```
; First, it sets up the sector, head and drive information.
; DS:000A holds the sector to read

135B:0126 880E0A00         MOV [000A],CL
135B:012A 8A160900         MOV DL,[0009]
135B:012E B600             MOV DH,00

; Sets the DTA

135B:0130 8D365802         LEA SI,[0258]
135B:0134 7213             JB 0149

; Resets the disk

135B:0136 33C0             XOR AX,AX
135B:0138 CD13             INT 13

; Calls the the check

135B:013A B90114           MOV CX,1401   ; TRACK 14 sector 1
135B:013D 8BDE             MOV BX,SI
135B:013F E8B300           CALL 01F5


; The next track/sector to read in is stored in BP

135B:0142 8BCD             MOV CX,BP
135B:0144 E8AE00           CALL 01F5
135B:0147 7234             JB 017D       ; If an error occured,
                                         ; trap it.


135B:0149 88160900         MOV [0009],DL     ; Reset drive
135B:014D 8A0E0A00         MOV CL,[000A]     ; reset sector
135B:0151 E8A100           CALL 01F5         ; check protection
135B:0154 722F             JB 0185           ; Check for an error

135B:0156 8D5C20           LEA BX,[SI+20]
```

```
135B:0159 8BCD          MOV CX,BP       ; Get next T/S
135B:015B B010          MOV AL,10       ; Ignore this
135B:015D E89500        CALL 01F5       ; Check protection
135B:0160 7223          JB 0185         ; check for error

; The next sector of code checks to see if what was read in
; is the actual protected tracks

; First check

135B:0162 8DBCAC00      LEA DI,[SI+00AC]
135B:0166 B91000        MOV CX,0010
135B:0169 F3            REPZ
135B:016A A7            CMPSW

; NOTE: If it was a bad track, it will jmp to 185.   A good
; read should just continue

135B:016B 7518          JNZ 0185

; Second check

135B:016D 8D365802      LEA SI,[0258]
135B:0171 8D3E3702      LEA DI,[0237]
135B:0175 B90400        MOV CX,0004
135B:0178 F3            REPZ
135B:0179 A7            CMPSW

; see NOTE above

135B:017A 7509          JNZ 0185

; This exit back to the main routine.

135B:017C C3            RET

; Here is the start of the error trap routines.   Basicly what
; they do is check an error count.   If it's not 0 then it
; retries everything.   If it is 0 then it exit back to dos.

135B:017D FEC2          INC DL
135B:017F 3A160200      CMP DL,[0002]
135B:0183 72B1          JB 0136
135B:0185 E85400        CALL 01DC
135B:0188 8B260B00      MOV SP,[000B]
135B:018C 2BC9          SUB CX,CX
135B:018E 58            POP AX
135B:018F 50            PUSH AX
135B:0190 EB1F          JMP 01B1


** Here is the actual code the does the check   **

; ES:BX points to the buffer
```

```
135B:01F5 1E                PUSH DS
135B:01F6 07                POP ES

; SI is set to the # of retries

135B:01F7 56                PUSH SI
135B:01F8 BE0600            MOV SI,0006

; Remember how I said we would use what was in DS:0000 later.
; well, here is where you use it.   It loads in the FUNCTION
; and # of sectors from what is stored in DS:0000.   This is
; just a trick to make the int 13 call more vague.

135B:01FB A10000            MOV AX,[0000]
135B:01FE CD13              INT 13

; If there is no errors, then exit this part of the loop

135B:0200 7309              JNB 020B
135B:0202 F6C480            TEST AH,80

; Check to see if it was a drive TIMEOUT.   If so, then set
; an error flag and exit

135B:0205 7503              JNZ 020A

; It must have been a load error.   Retry 6 times

135B:0207 4E                DEC SI
135B:0208 75F1              JNZ 01FB

; Set the error flag

135B:020A F9                STC

; restore SI and return

135B:020B 5E                POP SI
135B:020C C3                RET
```

If you follow through all of that.   You will see that the only real way out is the jmp to "135B:0192" at 135B:0119. So, how do we test it.   Simple.   Exit back to dos and let's add a temporary patch.

Reload ERE.COM under debug.   Execute the program setting a breakpoint at 135B:0022 (if you remember, that is right at the begining of the self modifing code).   When execution stops, change you IP register to 192.   Now execute the code.

Well shit, we are at the main menu.   We just bypassed the entire protection routine.   So, now where to add the patch.   We will be adding the patch at 135B:0022.   But what should the patch be.   In this case, simply jumping to 135B:0192 will do.   So, reload ERE.COM under debug.   Execute the code until 135B:0022. Now unassemble it. Here is the code fragment we need.

```
135B:0022 B013              MOV AL,13
135B:0024 A23500            MOV [0035],AL
135B:0027 A2FF01            MOV [01FF],AL
135B:002A A22F02            MOV [022F],AL
135B:002D A23901            MOV [0139],AL
```

Here is the code we want to use as the patch

```
135B:0022 E96D01            JMP 192
```

So, to add the patch, we search the file ERE.COM using PC-TOOLS. For our search string we use

```
     B0 13 A2 35 00 A2 FF 01 A2 2F 02 A2 39 01
```

PC-TOOLS should find the search string at reletive sector #13. Edit the sector and change "B0 13 A2" to "E9 6D 01" (our patch) and save the sector.

BOOM! your done and CSMP is cracked. Fun huh. You just kicked 5 seconds off of the load time. Preaty fucken good. Well, I hope this textfile helped.

-Buckaroo Banzai
 -Cracking Guru

CRACKING 101 - 1990 Edition

Lesson 4
revision 1

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ REMOVING THE DOC CHECK FOR STAR CONTROL ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

<

Added for revision 1 -

First, let me tell you about a major fuckup I made. When I first wrote this file, I left out a major part of the patch. For all of the user who got that version, I'm sorry but even I make mistakes at 3:00 in the morning. Anyway, just replace the original with this updated version

- Buckaroo Banzai

>

Hey, Buckaroo Banzai .. Cracking Guru back once again to help you lesser crackist learn. This time, we will be going over Star Control. This is the last lesson in the original 4. From here on out, I will simply release lessons as I write them.

I want to say a few things about some of the groups out there right now. Speed isn't everything. I really wish that for example when you remove a doc check, most of us want it REMOVED. We don't want to have to enter your group name or even typing 1 letter is to much. We shouldn't even see the menu for the doc check. Now, I don't direct this to all of you, but there seems to have been a move from quality to quickness. Let's go back to the days of SPI (and INC when they were first getting started) and crack right. If there is a doc check, remove it, not just fake it.

Nuff said, on with the tutorial.

Star Control (SC for here out) is a preaty good game. The protection on it wasn't too hard, but if you didn't read enough in to it, you would just kill the title music also.

So, how do we go about cracking SC. Well for this one I opted to break out when SC asks for the code from the code wheel. Originaly I did this just for the hell of it, but it turned out to be a luck guess and made life a lot easier.

As usual we will be using periscope to crack SC. I used PSKEY (using int 3 as the trap interrupt not int 2) to pop in at the input routine. So lets get started. Load up PS and PSKEY, then execute Star Control. When you get to the doc check, break out.

Now you should be at the usual IRET insturction that's part of PSKEY. Now comes the tricky part. Since we are using a key trap to break out during the input sequence, we could be anywhere inside the entire input routine. So in cases like this I suggest finding a reference point.

So how do you pick the reference point. Well, since this doc check must be entered via the keyboard you can bet somewhere it will call INT 16h (bios keyboard) (although there are times when this is not true, it rare). I think we should go off and find that call to that interrupt.

So we trace (using the 'T' command) through some code and finally come apon the follow subroutine ....

(NOTE: all comments were added by me)

```
; This is the actual routine that is used to get a key

2A00:09D4 55            PUSH BP
```

```
2A00:09D5 8BEC              MOV BP,SP
2A00:09D7 8A6606            MOV AH,[BP+06]
2A00:09DA 8AD4              MOV DL,AH
2A00:09DC 80E20F            AND DL,0F
2A00:09DF CD16              INT 16         ; Call to bios.   We will
2A00:09E1 7509              JNZ   09EC        ;   use   this   as   our
2A00:09E3 80FA01            CMP DL,01     ; reference point
2A00:09E6 7504              JNZ 09EC
2A00:09E8 33C0              XOR AX,AX
2A00:09EA EB0A              JMP 09F6
2A00:09EC 80FA02            CMP DL,02
2A00:09EF 7405              JZ 09F6
2A00:09F1 0BC0              OR AX,AX
2A00:09F3 7501              JNZ 09F6
2A00:09F5 48                DEC AX
2A00:09F6 5D                POP BP
2A00:09F7 CB                RETF
```

So we write down the address of our REFERENCE point and get ready to procede. Now, It's really kinda boring to keep trying to trace through the entire input routine while trying to enter the code string, so what we want to do next, is to figure out the input routine. A quick look at this last section of code shows that it only reads in a character but really does not handle it.

So, we exit via the RETF at 9F7 enter the next level of the subroutine. Again, if you manual trace through this routine (as well as the next level up) you see that it simple exits out rather quickly. This is definitly not the top loop of the imput routine.

So, we trace through the next level up, and again exit quickly to a higher level. But this time, as we trace through, we find that the it loops back on itself. AHA, the outer input loop. Here is the code to the entire input loop. I have marked the place where you should enter from the lower level.

(String input loop -- Outer level)

```
7C00:0835 FF365220          PUSH [2052]
7C00:0839 FF365020          PUSH [2050]
7C00:083D 9A2802FD41        CALL 41FD:0228         ;   Entery here
7C00:0842 888670FE          MOV [BP+FE70],AL
7C00:0946 0AC0              OR AL,AL
7C00:0848 7503              JNZ 084D
7C00:084A E99200            JMP 08DF
7C00:084D 2AE4              SUB AH,AH
7C00:084F 2D0800            SUB AX,0008
7C00:0852 745A              JZ 08AE
7C00:0854 48                DEC AX
7C00:0855 48                DEC AX
7C00:0856 7503              JNZ 085B
7C00:0858 E90901            JMP 0964
7C00:085B 2D0300            SUB AX,0003
```

```
7C00:085E 7503              JNZ 0863
7C00:0860 E90101            JMP 0964
7C00:0863 8A9E70FE          MOV BL,[BP+FE70]
7C00:0867 2AFF              SUB BH,BH
7C00:0869 F687790B57        TEST BYTE PTR [BX+0B79],57
7C00:086E 746F              JZ 08DF
7C00:0870 F687790B03        TEST BYTE PTR [BX+0B79],03
7C00:0875 740C              JZ 0883
7C00:0877 F687790B02        TEST BYTE PTR [BX+0B79],02
7C00:087C 7405              JZ 0883
7C00:087E 80AE70FE20        SUB BYTE PTR [BP+FE70],20
7C00:0883 8A8670FE          MOV AL,[BP+FE70]
7C00:0887 C49E7EFE          LES BX,[BP+FE7E]
7C00:088B 8BB682FE          MOV SI,[BP+FE82]
7C00:088F 26                ES:
7C00:0890 8800              MOV [BX+SI],AL
7C00:0892 FF8682FE          INC WORD PTR [BP+FE82]
7C00:0896 FFB688FE          PUSH [BP+FE88]
7C00:089A 8D8678FE          LEA AX,[BP+FE78]
7C00:089E 50                PUSH AX
7C00:089F 9A56049324        CALL 2493:0456
7C00:08A4 83C404            ADD SP,+04
7C00:08A7 0BC0              OR AX,AX
7C00:08A9 7534              JNZ 08DF
7C00:08AB EB27              JMP 08D4
7C00:08AD 90                NOP
7C00:08AE 83BE82FE00        CMP WORD PTR [BP+FE82],+00
7C00:08B3 7404              JZ 08B9
7C00:08B5 FF8E82FE          DEC WORD PTR [BP+FE82]
7C00:08B9 B008              MOV AL,08
7C00:08BB 50                PUSH AX
7C00:08BC 9A1003443D        CALL 3D44:0310
7C00:08C1 8D8684FE          LEA AX,[BP+FE84]
7C00:08C5 16                PUSH SS
7C00:08C6 50                PUSH AX
7C00:08C7 9A6A00843D        CALL 3D84:006A
7C00:08CC B047              MOV AL,47
7C00:08CE 50                PUSH AX
7C00:08CF 9A1003443D        CALL 3D44:0310
7C00:08D4 8D8678FE          LEA AX,[BP+FE78]
7C00:08D8 16                PUSH SS
7C00:08D9 50                PUSH AX
7C00:08DA 9A8202C93C        CALL 3CC9:0282
7C00:08DF 83BE8CFE00        CMP WORD PTR [BP+FE8C],+00
7C00:08E4 7503              JNZ 08E9
7C00:08E6 E94CFF            JMP 0835      ; <ÄÄÄ¿
                                            ³
```

as you can see, at this point it loops back on itself. This is what tells use that it's the outer loop. Knowing that, we can just set a code breakpoint at 8E9 (the next instruction after the loop) and execute the code.

At this point, the SC will pause waiting for you to enter the code key. Use the code wheel and enter the correct key (after all, it's kinda hard to crack a game without having the proper codes right...)

So, we have now exited the input loop with everything intact (ie: the proper code was entered). Next step is to figure out what happens when the proper code is entered. Well, since you have entered the proper code, just follow this routine out. Remember back to lesson 2. What we want to do is find the call the to routine that does the doc check and remove it somehow (a PROPER crack). So since everything is in the right place, if we just keep jumping over the code we should find our way out.

So after jumping over many instructions, we come the the follow piece of code

```
7C00:0B74 8BE5            MOV SP,BP
7C00:0B76 5D              POP BP
7C00:0B77 CB              RETF
```

By now, you should know that what you are looking at is the exit routine for a higher level language's (C or pascal) code. So we have found the end of the doc check. After tracing through the RETF you find yourself looking down a cmp and a conditional jump. Here is the code (NOTE! I have included the actual call to the doc check just for reference)

```
45E2:0235 9A46010F4A      CALL 7C00:146     ; Call to Doc Check
45E2:023A 83C404          ADD   SP,+04
45E2:023D 0BC0            OR    AX,AX
45E2:023F 7465            JZ    02A6
```

Notice the value of the AX register. Since right after the doc check, it is acted upon, then it has some importance. So, now that we know where the doc check takes place, how do we remove it.

Well, We could patch it with the code

```
45E2:0235 B40100          MOV   AX,0001
45E2:0238 90              NOP
45E2:0239 90              NOP
```

This patch will work (I know, it's how I first patched the program). But there is one small problem. If you run the program after adding this patch, you will find that the title music doesn't play. So, this is now a good place to put the patch.

So where then. Well, make note of the address of the call to the doc check. Now, restart the process but this time right after SC switches in to graphics mode, break out.

Now, set a breakpoint at the address from above (in my case 45E2:0235). Let SC run in to the intro. You will find that although the title screen comes up, the music doesn't kick in before the breakpoint is reached.

No, they couldn't...they wouldn't.. well they did. The music routines for the intro are stored in the routine for the doc check. Here is the entire doc check. I have commented on some of the code

```
; these first few calls seem to load something from disk


7C00:0146 55                PUSH BP
7C00:0147 8BEC              MOV BP,SP
7C00:0149 81EC9001          SUB SP,0190
7C00:014D 57                PUSH DI
7C00:014E 56                PUSH SI
7C00:014F 8B4608            MOV AX,[BP+08]
7C00:0152 0B4606            OR AX,[BP+06]
7C00:0155 740E              JZ 0165
7C00:0157 FF7608            PUSH [BP+08]
7C00:015A FF7606            PUSH [BP+06]
7C00:015D 9A65341E2D        CALL 2D1E:3465
7C00:0162 83C404            ADD SP,+04
7C00:0165 FF365220          PUSH [2052]
7C00:0169 FF365020          PUSH [2050]
7C00:016D 9A2802FD41        CALL 41FD:0228
7C00:0172 0AC0              OR AL,AL
7C00:0174 75EF              JNZ 0165
7C00:0176 B80200            MOV AX,0002
7C00:0179 898664FF          MOV [BP+FF64],AX
7C00:017D 898672FF          MOV [BP+FF72],AX
7C00:0181 2BC0              SUB AX,AX
7C00:0183 898662FF          MOV [BP+FF62],AX
7C00:0187 89866AFF          MOV [BP+FF6A],AX
7C00:018B 898674FF          MOV [BP+FF74],AX
7C00:018F B80100            MOV AX,0001
7C00:0192 898666FF          MOV [BP+FF66],AX
7C00:0196 89866CFF          MOV [BP+FF6C],AX
7C00:019A 898670FF          MOV [BP+FF70],AX
7C00:019E 898676FF          MOV [BP+FF76],AX
7C00:01A2 B80300            MOV AX,0003
7C00:01A5 898668FF          MOV [BP+FF68],AX
7C00:01A9 89866EFF          MOV [BP+FF6E],AX
7C00:01AD 898678FF          MOV [BP+FF78],AX

; Although I have NO IDEA what the hell is being setup
; here I suspect that it is the must

7C00:01B1 C746860400        MOV WORD PTR [BP-7A],0004
7C00:01B6 C746880100        MOV WORD PTR [BP-78],0001
7C00:01BB C7468A0200        MOV WORD PTR [BP-76],0002
7C00:01C0 C7468C0000        MOV WORD PTR [BP-74],0000
7C00:01C5 C7468E0000        MOV WORD PTR [BP-72],0000
7C00:01CA C746900500        MOV WORD PTR [BP-70],0005
7C00:01CF C746920600        MOV WORD PTR [BP-6E],0006
7C00:01D4 C746940700        MOV WORD PTR [BP-6C],0007
7C00:01D9 C746960C00        MOV WORD PTR [BP-6A],000C
7C00:01DE 894698            MOV [BP-68],AX
```

```
7C00:01E1 C7469A0500       MOV WORD PTR [BP-66],0005
7C00:01E6 C7469C0D00       MOV WORD PTR [BP-64],000D
7C00:01EB C7469E0000       MOV WORD PTR [BP-62],0000
7C00:01F0 C746A00100       MOV WORD PTR [BP-60],0001
7C00:01F5 C746A20200       MOV WORD PTR [BP-5E],0002
7C00:01FA C746A40800       MOV WORD PTR [BP-5C],0008
7C00:01FF B80400           MOV AX,0004
7C00:0202 8946A6           MOV [BP-5A],AX
7C00:0205 8946A8           MOV [BP-58],AX
7C00:0208 C746AA0600       MOV WORD PTR [BP-56],0006
7C00:020D C746AC0800       MOV WORD PTR [BP-54],0008
7C00:0212 C746AE0700       MOV WORD PTR [BP-52],0007
7C00:0217 C746B00900       MOV WORD PTR [BP-50],0009
7C00:021C C746B20A00       MOV WORD PTR [BP-4E],000A
7C00:0221 8946B4           MOV [BP-4C],AX
7C00:0224 C746B60C00       MOV WORD PTR [BP-4A],000C
7C00:0229 C746B80300       MOV WORD PTR [BP-48],0003
7C00:022E C746BA0B00       MOV WORD PTR [BP-46],000B
7C00:0233 C746BC0D00       MOV WORD PTR [BP-44],000D
7C00:0238 C746BE0B00       MOV WORD PTR [BP-42],000B
7C00:023D C746C00500       MOV WORD PTR [BP-40],0005
7C00:0242 C746C20100       MOV WORD PTR [BP-3E],0001
7C00:0247 C746C40700       MOV WORD PTR [BP-3C],0007
7C00:024C C746C60000       MOV WORD PTR [BP-3A],0000
7C00:0251 C746C80600       MOV WORD PTR [BP-38],0006
7C00:0256 C746CA0200       MOV WORD PTR [BP-36],0002
7C00:025B C746CC0300       MOV WORD PTR [BP-34],0003
7C00:0260 C746CE0800       MOV WORD PTR [BP-32],0008
7C00:0265 C746D00900       MOV WORD PTR [BP-30],0009
7C00:026A C746D20A00       MOV WORD PTR [BP-2E],000A
7C00:026F C746D40B00       MOV WORD PTR [BP-2C],000B
7C00:0274 C746D60C00       MOV WORD PTR [BP-2A],000C
7C00:0279 C746D80A00       MOV WORD PTR [BP-28],000A
7C00:027E C746DA0500       MOV WORD PTR [BP-26],0005
7C00:0283 C746DC0D00       MOV WORD PTR [BP-24],000D
7C00:0288 C746DE0800       MOV WORD PTR [BP-22],0008
7C00:028D C746E00900       MOV WORD PTR [BP-20],0009
7C00:0292 C746E20300       MOV WORD PTR [BP-1E],0003
7C00:0297 C746E40B00       MOV WORD PTR [BP-1C],000B
7C00:029C C78692FE0000   MOV WORD PTR [BP+FE92],0000
7C00:02A2 C78694FE2B00   MOV WORD PTR [BP+FE94],002B
7C00:02A8 C78696FE0200   MOV WORD PTR [BP+FE96],0002
7C00:02AE C78698FE0300   MOV WORD PTR [BP+FE98],0003
7C00:02B4 89869AFE         MOV [BP+FE9A],AX
7C00:02B8 C7869CFE0500   MOV WORD PTR [BP+FE9C],0005
7C00:02BE C7869EFE0600   MOV WORD PTR [BP+FE9E],0006
7C00:02C4 C786A0FE0E00   MOV WORD PTR [BP+FEA0],000E
7C00:02CA C786A2FE2B00   MOV WORD PTR [BP+FEA2],002B
7C00:02D0 C786A4FE0900   MOV WORD PTR [BP+FEA4],0009
7C00:02D6 C786A6FE0A00   MOV WORD PTR [BP+FEA6],000A
7C00:02DC C786A8FE0B00   MOV WORD PTR [BP+FEA8],000B
7C00:02E2 C786AAFE0C00   MOV WORD PTR [BP+FEAA],000C
7C00:02E8 C786ACFE2B00   MOV WORD PTR [BP+FEAC],002B
7C00:02EE C786AEFE0F00   MOV WORD PTR [BP+FEAE],000F
7C00:02F4 C786B0FE0D00   MOV WORD PTR [BP+FEB0],000D
7C00:02FA C786B2FE1000   MOV WORD PTR [BP+FEB2],0010
```

```
7C00:0300 C786B4FE1100   MOV WORD PTR [BP+FEB4],0011
7C00:0306 C786B6FE1200   MOV WORD PTR [BP+FEB6],0012
7C00:030C C786B8FE1300   MOV WORD PTR [BP+FEB8],0013
7C00:0312 C786BAFE1400   MOV WORD PTR [BP+FEBA],0014
7C00:0318 C786BCFE1500   MOV WORD PTR [BP+FEBC],0015
7C00:031E C786BEFE1600   MOV WORD PTR [BP+FEBE],0016
7C00:0324 C786C0FE1700   MOV WORD PTR [BP+FEC0],0017
7C00:032A C786C2FE0800   MOV WORD PTR [BP+FEC2],0008
7C00:0330 C786C4FE1800   MOV WORD PTR [BP+FEC4],0018
7C00:0336 C786C6FE2B00   MOV WORD PTR [BP+FEC6],002B
7C00:033C C786C8FE1900   MOV WORD PTR [BP+FEC8],0019
7C00:0342 C786CAFE2B00   MOV WORD PTR [BP+FECA],002B
7C00:0348 C786CCFE1A00   MOV WORD PTR [BP+FECC],001A
7C00:034E C786CEFE1B00   MOV WORD PTR [BP+FECE],001B
7C00:0354 C786D0FE1C00   MOV WORD PTR [BP+FED0],001C
7C00:035A C786D2FE1D00   MOV WORD PTR [BP+FED2],001D
7C00:0360 C786D4FE1E00   MOV WORD PTR [BP+FED4],001E
7C00:0366 C786D6FE1F00   MOV WORD PTR [BP+FED6],001F
7C00:036C C786D8FE2000   MOV WORD PTR [BP+FED8],0020
7C00:0372 C786DAFE2100   MOV WORD PTR [BP+FEDA],0021
7C00:0378 C786DCFE0700   MOV WORD PTR [BP+FEDC],0007
7C00:037E C786DEFE2200   MOV WORD PTR [BP+FEDE],0022
7C00:0384 C786E0FE2300   MOV WORD PTR [BP+FEE0],0023
7C00:038A C786E2FE2400   MOV WORD PTR [BP+FEE2],0024
7C00:0390 C786E4FE2500   MOV WORD PTR [BP+FEE4],0025
7C00:0396 C786E6FE2600   MOV WORD PTR [BP+FEE6],0026
7C00:039C C786E8FE2B00   MOV WORD PTR [BP+FEE8],002B
7C00:03A2 C786EAFE2700   MOV WORD PTR [BP+FEEA],0027
7C00:03A8 C786ECFE2800   MOV WORD PTR [BP+FEEC],0028
7C00:03AE C786EEFE2900   MOV WORD PTR [BP+FEEE],0029
7C00:03B4 C786F0FE2A00   MOV WORD PTR [BP+FEF0],002A
7C00:03BA 8D46F4         LEA AX,[BP-0C]
7C00:03BD 50             PUSH AX
7C00:03BE 8D867AFF       LEA AX,[BP+FF7A]
7C00:03C2 50             PUSH AX
7C00:03C3 8D862CFF       LEA AX,[BP+FF2C]
7C00:03C7 50             PUSH AX
7C00:03C8 8D8628FF       LEA AX,[BP+FF28]
7C00:03CC 50             PUSH AX
7C00:03CD E832FC         CALL 0002     ; Music Plays
7C00:03D0 0BC0           OR AX,AX
7C00:03D2 7503           JNZ 03D7
7C00:03D4 E99B07         JMP 0B72
7C00:03D7 FF36AA1E       PUSH [1EAA]
7C00:03DB 9A0200443D     CALL 3D44:0002
7C00:03E0 FF36AE1E       PUSH [1EAE]
7C00:03E4 FF36AC1E       PUSH [1EAC]
7C00:03E8 9A0C008D3D     CALL 3D8D:000C
7C00:03ED B80201         MOV AX,0102
7C00:03F0 50             PUSH AX
7C00:03F1 9ADE02443D     CALL 3D44:02DE
7C00:03F6 B80400         MOV AX,0004
7C00:03F9 BA4000         MOV DX,0040
7C00:03FC 52             PUSH DX
7C00:03FD 50             PUSH AX
7C00:03FE 8D868CFE       LEA AX,[BP+FE8C]
```

```
7C00:0402 50              PUSH AX
7C00:0403 9A7000963B      CALL 3B96:0070       ; Music plays
7C00:0408 89868EFE        MOV [BP+FE8E],AX
7C00:040C 899690FE        MOV [BP+FE90],DX
7C00:0410 0BD0            OR DX,AX
7C00:0412 7471            JZ 0485
7C00:0414 2BC0            SUB AX,AX
7C00:0416 898686FE        MOV [BP+FE86],AX
7C00:041A 898684FE        MOV [BP+FE84],AX
7C00:041E FFB690FE        PUSH [BP+FE90]
7C00:0422 FFB68EFE        PUSH [BP+FE8E]
7C00:0426 9A0A00F93C      CALL 3CF9:000A
7C00:042B 898688FE        MOV [BP+FE88],AX
7C00:042F 89968AFE        MOV [BP+FE8A],DX
7C00:0433 833EB41E00      CMP WORD PTR [1EB4],+00
7C00:0438 7514            JNZ 044E
7C00:043A 8B4608          MOV AX,[BP+08]
7C00:043D 0B4606          OR AX,[BP+06]
7C00:0440 740C            JZ 044E
7C00:0442 B80100          MOV AX,0001
7C00:0445 50              PUSH AX
7C00:0446 9AF4019324      CALL 2493:01F4
7C00:044B 83C402          ADD SP,+02
7C00:044E 2AC0            SUB AL,AL
7C00:0450 50              PUSH AX
7C00:0451 9A4803443D      CALL 3D44:0348
7C00:0456 9A57331E2D      CALL 2D1E:3357
7C00:045B 9A9911A73B      CALL 3BA7:1199
7C00:0460 8D8684FE        LEA AX,[BP+FE84]
7C00:0464 16              PUSH SS
7C00:0465 50              PUSH AX
7C00:0466 9A04007E3D      CALL 3D7E:0004       ; Music plays
7C00:046B FFB68AFE        PUSH [BP+FE8A]
7C00:046F FFB688FE        PUSH [BP+FE88]
7C00:0473 9AF001F93C      CALL 3CF9:01F0
7C00:0478 FFB690FE        PUSH [BP+FE90]
7C00:047C FFB68EFE        PUSH [BP+FE8E]
7C00:0480 9A78068D3D      CALL 3D8D:0678       ; Music plays
7C00:0485 8B4608          MOV AX,[BP+08]
7C00:0488 0B4606          OR AX,[BP+06]
7C00:048B 7429            JZ 04B6
7C00:048D 833EB41E00      CMP WORD PTR [1EB4],+00
7C00:0492 740C            JZ 04A0
7C00:0494 B80100          MOV AX,0001
7C00:0497 50              PUSH AX
7C00:0498 9AF4019324      CALL 2493:01F4       ; Music Plays
7C00:049D 83C402          ADD SP,+02
7C00:04A0 9A8C341E2D      CALL 2D1E:348C
7C00:04A5 FF7608          PUSH [BP+08]
7C00:04A8 FF7606          PUSH [BP+06]
7C00:04AB 9A2A006342      CALL 4263:002A
7C00:04B0 50              PUSH AX
7C00:04B1 9A54006342      CALL 4263:0054

; this is the start of the actual   doc check.   OH! As you can
; tell, I wasn't too intrested in the music routines, but
; thought it might be fun to track them down

7C00:04B6 9AD0098D3D      CALL 3D8D:09D0   ; Show Doc check
                                           ; screen
7C00:04BB B80301          MOV AX,0103
7C00:04BE 50              PUSH AX
7C00:04BF 9ADE02443D      CALL 3D44:02DE
7C00:04C4 C746F60B00      MOV WORD PTR [BP-0A],000B
7C00:04C9 C746F87900      MOV WORD PTR [BP-08],0079
7C00:04CE C746FA2801      MOV WORD PTR [BP-06],0128
7C00:04D3 C746FC4500      MOV WORD PTR [BP-04],0045
7C00:04D8 B008            MOV AL,08
7C00:04DA 50              PUSH AX
7C00:04DB 9A1003443D      CALL 3D44:0310
7C00:04E0 8D867AFF        LEA AX,[BP+FF7A]
7C00:04E4 16              PUSH SS
7C00:04E5 50              PUSH AX
7C00:04E6 9A36007E3D      CALL 3D7E:0036   ; Show alien's face

7C00:04EB C746E6A000      MOV WORD PTR [BP-1A],00A0
7C00:04F0 C746EA0100      MOV WORD PTR [BP-16],0001
7C00:04F5 C746840300      MOV WORD PTR [BP-7C],0003
7C00:04FA 2AC0            SUB AL,AL
7C00:04FC 50              PUSH AX
7C00:04FD 9A1003443D      CALL 3D44:0310
7C00:0502 8B46F8          MOV AX,[BP-08]
7C00:0505 050700          ADD AX,0007
7C00:0508 8946E8          MOV [BP-18],AX
7C00:050B FFB62EFF        PUSH [BP+FF2E]
7C00:050F FFB62CFF        PUSH [BP+FF2C]
7C00:0513 FFB62EFF        PUSH [BP+FF2E]
7C00:0517 FFB62CFF        PUSH [BP+FF2C]
7C00:051B 9AE400FC44      CALL 44FC:00E4
7C00:0520 8BF0            MOV SI,AX
7C00:0522 9A1201E245      CALL 45E2:0112
7C00:0527 B90500          MOV CX,0005
7C00:052A 8BD0            MOV DX,AX
7C00:052C 8BC6            MOV AX,SI
7C00:052E 8BDA            MOV BX,DX
7C00:0530 2BD2            SUB DX,DX
7C00:0532 F7F1            DIV CX
7C00:0534 8BD0            MOV DX,AX
7C00:0536 4A              DEC DX
7C00:0537 8BC3            MOV AX,BX
7C00:0539 8BDA            MOV BX,DX
7C00:053B 2BD2            SUB DX,DX
7C00:053D F7F3            DIV BX
7C00:053F 42              INC DX
7C00:0540 8BC2            MOV AX,DX
7C00:0542 D1E2            SHL DX,1
7C00:0544 D1E2            SHL DX,1
7C00:0546 03D0            ADD DX,AX
7C00:0548 52              PUSH DX
7C00:0549 9A2801FC44      CALL 44FC:0128
7C00:054E 89868EFE        MOV [BP+FE8E],AX
7C00:0552 899690FE        MOV [BP+FE90],DX
```

```
7C00:0556 C78672FE0000   MOV WORD PTR [BP+FE72],0000

; This is the start of the loop the prints out the stupid
; message

7C00:055C 52                PUSH DX
7C00:055D 50                PUSH AX
7C00:055E 9A4602FC44        CALL 44FC:0246
7C00:0563 8946EC            MOV [BP-14],AX
7C00:0566 8956EE            MOV [BP-12],DX
7C00:0569 FFB690FE          PUSH [BP+FE90]
7C00:056D FFB68EFE          PUSH [BP+FE8E]
7C00:0571 9AF201FC44        CALL 44FC:01F2
7C00:0576 8946F0            MOV [BP-10],AX
7C00:0579 8D46E6            LEA AX,[BP-1A]
7C00:057C 16                PUSH SS
7C00:057D 50                PUSH AX
7C00:057E 9A8202C93C        CALL 3CC9:0282
7C00:0583 8346E80A          ADD WORD PTR [BP-18],+0A
7C00:0587 FFB690FE          PUSH [BP+FE90]
7C00:058B FFB68EFE          PUSH [BP+FE8E]
7C00:058F B80100            MOV AX,0001
7C00:0592 50                PUSH AX
7C00:0593 9A7E01FC44        CALL 44FC:017E
7C00:0598 89868EFE          MOV [BP+FE8E],AX
7C00:059C 899690FE          MOV [BP+FE90],DX
7C00:05A0 FF8672FE          INC WORD PTR [BP+FE72]
7C00:05A4 83BE72FE05        CMP WORD PTR [BP+FE72],+05
7C00:05A9 7CB1              JL 055C

; Reads in the code to check   (I think.   Oh hell it really
; doesn't matter)

7C00:05AB 9A1201E245        CALL 45E2:0112
7C00:05B0 B90C00            MOV CX,000C
7C00:05B3 99                CWD
7C00:05B4 F7F9              IDIV CX
7C00:05B6 895682            MOV [BP-7E],DX
7C00:05B9 9A1201E245        CALL 45E2:0112
7C00:05BE B90C00            MOV CX,000C
7C00:05C1 99                CWD
7C00:05C2 F7F9              IDIV CX
7C00:05C4 8956F2            MOV [BP-0E],DX
7C00:05C7 9A1201E245        CALL 45E2:0112
7C00:05CC B90C00            MOV CX,000C
7C00:05CF 99                CWD
7C00:05D0 F7F9              IDIV CX
7C00:05D2 8956FE            MOV [BP-02],DX
7C00:05D5 9A1201E245        CALL 45E2:0112
7C00:05DA B90C00            MOV CX,000C
7C00:05DD 99                CWD
7C00:05DE F7F9              IDIV CX
7C00:05E0 8996F4FE          MOV [BP+FEF4],DX
7C00:05E4 FFB62AFF          PUSH [BP+FF2A]
7C00:05E8 FFB628FF          PUSH [BP+FF28]
7C00:05EC FF7682            PUSH [BP-7E]
```

```
7C00:05EF 9A2801FC44       CALL 44FC:0128
7C00:05F4 89868EFE         MOV [BP+FE8E],AX
7C00:05F8 899690FE         MOV [BP+FE90],DX
7C00:05FC 52               PUSH DX
7C00:05FD 50               PUSH AX
7C00:05FE 8D86F6FE         LEA AX,[BP+FEF6]
7C00:0602 16               PUSH SS
7C00:0603 50               PUSH AX
7C00:0604 9A9A02FC44       CALL 44FC:029A
7C00:0609 FFB62AFF         PUSH [BP+FF2A]
7C00:060D FFB628FF         PUSH [BP+FF28]
7C00:0611 8B46FE           MOV AX,[BP-02]
7C00:0614 050C00           ADD AX,000C
7C00:0617 50               PUSH AX
7C00:0618 9A2801FC44       CALL 44FC:0128
7C00:061D 89868EFE         MOV [BP+FE8E],AX
7C00:0621 899690FE         MOV [BP+FE90],DX
7C00:0625 52               PUSH DX
7C00:0626 50               PUSH AX
7C00:0627 8DBEF6FE         LEA DI,[BP+FEF6]
7C00:062B 16               PUSH SS
7C00:062C 07               POP ES
7C00:062D B9FFFF           MOV CX,FFFF
7C00:0630 33C0             XOR AX,AX
7C00:0632 F2               REPNZ
7C00:0633 AE               SCASB
7C00:0634 F7D1             NOT CX
7C00:0636 49               DEC CX
7C00:0637 8BF1             MOV SI,CX
7C00:0639 8D82F6FE         LEA AX,[BP+SI+FEF6]
7C00:063D 16               PUSH SS
7C00:063E 50               PUSH AX
7C00:063F 9A9A02FC44       CALL 44FC:029A
7C00:0644 FFB62AFF         PUSH [BP+FF2A]
7C00:0648 FFB628FF         PUSH [BP+FF28]
7C00:064C 8B46F2           MOV AX,[BP-0E]
7C00:064F 051800           ADD AX,0018
7C00:0652 50               PUSH AX
7C00:0653 9A2801FC44       CALL 44FC:0128
7C00:0658 89868EFE         MOV [BP+FE8E],AX
7C00:065C 899690FE         MOV [BP+FE90],DX
7C00:0660 52               PUSH DX
7C00:0661 50               PUSH AX
7C00:0662 8DBEF6FE         LEA DI,[BP+FEF6]
7C00:0666 16               PUSH SS
7C00:0667 07               POP ES
7C00:0668 B9FFFF           MOV CX,FFFF
7C00:066B 33C0             XOR AX,AX
7C00:066D F2               REPNZ
7C00:066E AE               SCASB
7C00:066F F7D1             NOT CX
7C00:0671 49               DEC CX
7C00:0672 8BF1             MOV SI,CX
7C00:0674 8D82F6FE         LEA AX,[BP+SI+FEF6]
7C00:0678 16               PUSH SS
7C00:0679 50               PUSH AX
```

```
7C00:067A 9A9A02FC44       CALL 44FC:029A
7C00:067F FFB62AFF         PUSH [BP+FF2A]
7C00:0683 FFB628FF         PUSH [BP+FF28]
7C00:0687 8B86F4FE         MOV AX,[BP+FEF4]
7C00:068B 052400           ADD AX,0024
7C00:068E 50               PUSH AX
7C00:068F 9A2801FC44       CALL 44FC:0128
7C00:0694 89868EFE         MOV [BP+FE8E],AX
7C00:0698 899690FE         MOV [BP+FE90],DX
7C00:069C 52               PUSH DX
7C00:069D 50               PUSH AX
7C00:069E 8DBEF6FE         LEA DI,[BP+FEF6]
7C00:06A2 16               PUSH SS
7C00:06A3 07               POP ES
7C00:06A4 B9FFFF           MOV CX,FFFF
7C00:06A7 33C0             XOR AX,AX
7C00:06A9 F2               REPNZ
7C00:06AA AE               SCASB
7C00:06AB F7D1             NOT CX
7C00:06AD 49               DEC CX
7C00:06AE 8BF1             MOV SI,CX
7C00:06B0 8D82F6FE         LEA AX,[BP+SI+FEF6]
7C00:06B4 16               PUSH SS
7C00:06B5 50               PUSH AX
7C00:06B6 9A9A02FC44       CALL 44FC:029A
7C00:06BB C746E8B200       MOV WORD PTR [BP-18],00B2
7C00:06C0 8D86F6FE         LEA AX,[BP+FEF6]
7C00:06C4 8946EC           MOV [BP-14],AX
7C00:06C7 8C56EE           MOV [BP-12],SS
7C00:06CA 8DBEF6FE         LEA DI,[BP+FEF6]
7C00:06CE 16               PUSH SS
7C00:06CF 07               POP ES
7C00:06D0 B9FFFF           MOV CX,FFFF
7C00:06D3 33C0             XOR AX,AX
7C00:06D5 F2               REPNZ
7C00:06D6 AE               SCASB
7C00:06D7 F7D1             NOT CX
7C00:06D9 49               DEC CX
7C00:06DA 894EF0           MOV [BP-10],CX
7C00:06DD B084             MOV AL,84
7C00:06DF 50               PUSH AX
7C00:06E0 9A1003443D       CALL 3D44:0310
7C00:06E5 8D46E6           LEA AX,[BP-1A]
7C00:06E8 16               PUSH SS
7C00:06E9 50               PUSH AX
7C00:06EA 9A8202C93C       CALL 3CC9:0282    ; Displays the code
                                             ; to check

7C00:06EF 8346E80A         ADD WORD PTR [BP-18],+0A
7C00:06F3 FFB62AFF         PUSH [BP+FF2A]
7C00:06F7 FFB628FF         PUSH [BP+FF28]
7C00:06FB B85B00           MOV AX,005B
7C00:06FE 50               PUSH AX
7C00:06FF 9A2801FC44       CALL 44FC:0128
7C00:0704 89868EFE         MOV [BP+FE8E],AX
7C00:0708 899690FE         MOV [BP+FE90],DX
```

```
7C00:070C 52              PUSH DX
7C00:070D 50              PUSH AX
7C00:070E 9A4602FC44      CALL 44FC:0246
7C00:0713 8946EC          MOV [BP-14],AX
7C00:0716 8956EE          MOV [BP-12],DX
7C00:0719 FFB690FE        PUSH [BP+FE90]
7C00:071D FFB68EFE        PUSH [BP+FE8E]
7C00:0721 9AF201FC44      CALL 44FC:01F2
7C00:0726 8946F0          MOV [BP-10],AX
7C00:0729 2AC0            SUB AL,AL
7C00:072B 50              PUSH AX
7C00:072C 9A1003443D      CALL 3D44:0310
7C00:0731 8D46E6          LEA AX,[BP-1A]
7C00:0734 16              PUSH SS
7C00:0735 50              PUSH AX
7C00:0736 9A8202C93C      CALL 3CC9:0282    ; Displays "PROPER
                                            ; response" msg

7C00:073B 8B86F4FE        MOV AX,[BP+FEF4]
7C00:073F 2B46F2          SUB AX,[BP-0E]
7C00:0742 898672FE        MOV [BP+FE72],AX
7C00:0746 0346FE          ADD AX,[BP-02]
7C00:0749 898676FE        MOV [BP+FE76],AX
7C00:074D 0BC0            OR AX,AX
7C00:074F 7D09            JGE 075A
7C00:0751 050C00          ADD AX,000C
7C00:0754 898676FE        MOV [BP+FE76],AX
7C00:0758 EB0A            JMP 0764
7C00:075A 3D0C00          CMP AX,000C
7C00:075D 7C05            JL 0764
7C00:075F 83AE76FE0C      SUB WORD PTR [BP+FE76],+0C
7C00:0764 8B4682          MOV AX,[BP-7E]
7C00:0767 038672FE        ADD AX,[BP+FE72]
7C00:076B 898674FE        MOV [BP+FE74],AX
7C00:076F 0BC0            OR AX,AX
7C00:0771 7D09            JGE 077C
7C00:0773 050C00          ADD AX,000C
7C00:0776 898674FE        MOV [BP+FE74],AX
7C00:077A EB0A            JMP 0786
7C00:077C 3D0C00          CMP AX,000C
7C00:077F 7C05            JL 0786
7C00:0781 83AE74FE0C      SUB WORD PTR [BP+FE74],+0C
7C00:0786 8BB6F4FE        MOV SI,[BP+FEF4]
7C00:078A D1E6            SHL SI,1
7C00:078C 8BB262FF        MOV SI,[BP+SI+FF62]
7C00:0790 89B672FE        MOV [BP+FE72],SI
7C00:0794 8B8676FE        MOV AX,[BP+FE76]
7C00:0798 D1E0            SHL AX,1
7C00:079A D1E0            SHL AX,1
7C00:079C 03F0            ADD SI,AX
7C00:079E D1E6            SHL SI,1
7C00:07A0 8B8292FE        MOV AX,[BP+SI+FE92]
7C00:07A4 8986F4FE        MOV [BP+FEF4],AX
7C00:07A8 3D2B00          CMP AX,002B
7C00:07AB 7515            JNZ 07C2
7C00:07AD 8BB674FE        MOV SI,[BP+FE74]
```

```
7C00:07B1 D1E6              SHL SI,1
7C00:07B3 D1E6              SHL SI,1
7C00:07B5 03B672FE          ADD SI,[BP+FE72]
7C00:07B9 D1E6              SHL SI,1
7C00:07BB 8B4286            MOV AX,[BP+SI-7A]
7C00:07BE 8986F4FE          MOV [BP+FEF4],AX
7C00:07C2 C78684FE7800      MOV WORD PTR [BP+FE84],0078
7C00:07C8 B85100            MOV AX,0051
7C00:07CB 898686FE          MOV [BP+FE86],AX
7C00:07CF 898688FE          MOV [BP+FE88],AX
7C00:07D3 C7868AFE0900      MOV WORD PTR [BP+FE8A],0009
7C00:07D9 C78678FE7900      MOV WORD PTR [BP+FE78],0079
7C00:07DF C7867AFE5900      MOV WORD PTR [BP+FE7A],0059
7C00:07E5 C7867CFE0000      MOV WORD PTR [BP+FE7C],0000
7C00:07EB 8D86F6FE          LEA AX,[BP+FEF6]
7C00:07EF 89867EFE          MOV [BP+FE7E],AX
7C00:07F3 8C9680FE          MOV [BP+FE80],SS
7C00:07F7 C78682FE0000      MOV WORD PTR [BP+FE82],0000
7C00:07FD FFB62AFF          PUSH [BP+FF2A]
7C00:0801 FFB628FF          PUSH [BP+FF28]
7C00:0805 8B86F4FE          MOV AX,[BP+FEF4]
7C00:0809 053000            ADD AX,0030
7C00:080C 50                PUSH AX
7C00:080D 9A2801FC44        CALL 44FC:0128
7C00:0812 89868EFE          MOV [BP+FE8E],AX
7C00:0816 899690FE          MOV [BP+FE90],DX
7C00:081A 52                PUSH DX
7C00:081B 50                PUSH AX
7C00:081C 8D8630FF          LEA AX,[BP+FF30]
7C00:0820 16                PUSH SS
7C00:0821 50                PUSH AX
7C00:0822 9A9A02FC44        CALL 44FC:029A
7C00:0827 B047              MOV AL,47
7C00:0829 50                PUSH AX
7C00:082A 9A1003443D        CALL 3D44:0310
7C00:082F C7868CFE0000      MOV WORD PTR [BP+FE8C],0000

; All the code you just saw.   I have no clue what it does
; (hey at least I'm honest) but it wasn't important.

; Here is the imput outer loop

7C00:0835 FF365220          PUSH [2052]
7C00:0839 FF365020          PUSH [2050]
7C00:083D 9A2802FD41        CALL 41FD:0228
7C00:0842 888670FE          MOV [BP+FE70],AL
7C00:0846 0AC0              OR AL,AL
7C00:0848 7503              JNZ 084D
7C00:084A E99200            JMP 08DF
7C00:084D 2AE4              SUB AH,AH
7C00:084F 2D0800            SUB AX,0008
7C00:0852 745A              JZ 08AE
7C00:0854 48                DEC AX
7C00:0855 48                DEC AX
7C00:0856 7503              JNZ 085B
7C00:0858 E90901            JMP 0964
```

```
7C00:085B 2D0300            SUB AX,0003
7C00:085E 7503              JNZ 0863
7C00:0860 E90101            JMP 0964
7C00:0863 8A9E70FE          MOV BL,[BP+FE70]
7C00:0867 2AFF              SUB BH,BH
7C00:0869 F687790B57        TEST BYTE PTR [BX+0B79],57
7C00:086E 746F              JZ 08DF
7C00:0870 F687790B03        TEST BYTE PTR [BX+0B79],03
7C00:0875 740C              JZ 0883
7C00:0877 F687790B02        TEST BYTE PTR [BX+0B79],02
7C00:087C 7405              JZ 0883
7C00:087E 80AE70FE20        SUB BYTE PTR [BP+FE70],20
7C00:0883 8A8670FE          MOV AL,[BP+FE70]
7C00:0887 C49E7EFE          LES BX,[BP+FE7E]
7C00:088B 8BB682FE          MOV SI,[BP+FE82]
7C00:088F 26                ES:
7C00:0890 8800              MOV [BX+SI],AL
7C00:0892 FF8682FE          INC WORD PTR [BP+FE82]
7C00:0896 FFB688FE          PUSH [BP+FE88]
7C00:089A 8D8678FE          LEA AX,[BP+FE78]
7C00:089E 50                PUSH AX
7C00:089F 9A56049324        CALL 2493:0456
7C00:08A4 83C404            ADD SP,+04
7C00:08A7 0BC0              OR AX,AX
7C00:08A9 7534              JNZ 08DF
7C00:08AB EB27              JMP 08D4
7C00:08AD 90                NOP
7C00:08AE 83BE82FE00        CMP WORD PTR [BP+FE82],+00
7C00:08B3 7404              JZ 08B9
7C00:08B5 FF8E82FE          DEC WORD PTR [BP+FE82]
7C00:08B9 B008              MOV AL,08
7C00:08BB 50                PUSH AX
7C00:08BC 9A1003443D        CALL 3D44:0310
7C00:08C1 8D8684FE          LEA AX,[BP+FE84]
7C00:08C5 16                PUSH SS
7C00:08C6 50                PUSH AX
7C00:08C7 9A6A00843D        CALL 3D84:006A
7C00:08CC B047              MOV AL,47
7C00:08CE 50                PUSH AX
7C00:08CF 9A1003443D        CALL 3D44:0310
7C00:08D4 8D8678FE          LEA AX,[BP+FE78]
7C00:08D8 16                PUSH SS
7C00:08D9 50                PUSH AX
7C00:08DA 9A8202C93C        CALL 3CC9:0282
7C00:08DF 83BE8CFE00        CMP WORD PTR [BP+FE8C],+00
7C00:08E4 7503              JNZ 08E9
7C00:08E6 E94CFF            JMP 0835

; Next comes the code that checks your entry.   If you follow
; it through you will see it handles not only clearing the
; screen and printing the "GOOD GOING" message but it also
; handles bad entries, etc.

7C00:08E9 8BB682FE          MOV SI,[BP+FE82]
7C00:08ED C682F6FE00        MOV BYTE PTR [BP+SI+FEF6],00
7C00:08F2 8DBE30FF          LEA DI,[BP+FF30]
```

```
7C00:08F6 8DB6F6FE        LEA SI,[BP+FEF6]
7C00:08FA 16              PUSH SS
7C00:08FB 07              POP ES
7C00:08FC B9FFFF          MOV CX,FFFF
7C00:08FF 33C0            XOR AX,AX
7C00:0901 F2              REPNZ
7C00:0902 AE              SCASB
7C00:0903 F7D1            NOT CX
7C00:0905 2BF9            SUB DI,CX
7C00:0907 F3              REPZ
7C00:0908 A6              CMPSB
7C00:0909 7405            JZ 0910
7C00:090B 1BC0            SBB AX,AX
7C00:090D 1DFFFF          SBB AX,FFFF
7C00:0910 3D0100          CMP AX,0001
7C00:0913 1BC0            SBB AX,AX
7C00:0915 F7D8            NEG AX
7C00:0917 8986F2FE        MOV [BP+FEF2],AX
7C00:091B 0BC0            OR AX,AX
7C00:091D 7509            JNZ 0928
7C00:091F 837E8401        CMP WORD PTR [BP-7C],+01
7C00:0923 7703            JA 0928
7C00:0925 E91C02          JMP 0B44
7C00:0928 0BC0            OR AX,AX
7C00:092A 7506            JNZ 0932
7C00:092C 837E8403        CMP WORD PTR [BP-7C],+03
7C00:0930 740A            JZ 093C
7C00:0932 0BC0            OR AX,AX
7C00:0934 745E            JZ 0994
7C00:0936 837E8403        CMP WORD PTR [BP-7C],+03
7C00:093A 7358            JNB 0994
7C00:093C B047            MOV AL,47
7C00:093E 50              PUSH AX
7C00:093F 9A1003443D      CALL 3D44:0310
7C00:0944 8D867AFF        LEA AX,[BP+FF7A]
7C00:0948 16              PUSH SS
7C00:0949 50              PUSH AX
7C00:094A 9A36007E3D      CALL 3D7E:0036
7C00:094F 83BEF2FE00      CMP WORD PTR [BP+FEF2],+00
7C00:0954 7518            JNZ 096E
7C00:0956 FF7680          PUSH [BP-80]
7C00:0959 FFB67EFF        PUSH [BP+FF7E]
7C00:095D 9A1C04F93C      CALL 3CF9:041C
7C00:0962 EB16            JMP 097A
7C00:0964 C7868CFE0100    MOV WORD PTR [BP+FE8C],0001
7C00:096A E972FF          JMP 08DF
7C00:096D 90              NOP
7C00:096E FF7680          PUSH [BP-80]
7C00:0971 FFB67EFF        PUSH [BP+FF7E]
7C00:0975 9A7204F93C      CALL 3CF9:0472
7C00:097A 89867EFF        MOV [BP+FF7E],AX
7C00:097E 895680          MOV [BP-80],DX
7C00:0981 B008            MOV AL,08
7C00:0983 50              PUSH AX
7C00:0984 9A1003443D      CALL 3D44:0310
7C00:0989 8D867AFF        LEA AX,[BP+FF7A]
```

```
7C00:098D 16              PUSH SS
7C00:098E 50              PUSH AX
7C00:098F 9A36007E3D      CALL 3D7E:0036
7C00:0994 B047            MOV AL,47
7C00:0996 50              PUSH AX
7C00:0997 9A1003443D      CALL 3D44:0310
7C00:099C 8D46F6          LEA AX,[BP-0A]
7C00:099F 16              PUSH SS
7C00:09A0 50              PUSH AX
7C00:09A1 9A6A00843D      CALL 3D84:006A
7C00:09A6 B008            MOV AL,08
7C00:09A8 50              PUSH AX
7C00:09A9 9A1003443D      CALL 3D44:0310
7C00:09AE 8D8684FE        LEA AX,[BP+FE84]
7C00:09B2 16              PUSH SS
7C00:09B3 50              PUSH AX
7C00:09B4 9A6A00843D      CALL 3D84:006A
7C00:09B9 83BEF2FE00      CMP WORD PTR [BP+FEF2],+00
7C00:09BE 7503            JNZ 09C3
7C00:09C0 E98500          JMP 0A48
7C00:09C3 2AC0            SUB AL,AL
7C00:09C5 50              PUSH AX
7C00:09C6 9A1003443D      CALL 3D44:0310
7C00:09CB 8B46F8          MOV AX,[BP-08]
7C00:09CE 050700          ADD AX,0007
7C00:09D1 8946E8          MOV [BP-18],AX
7C00:09D4 FFB62EFF        PUSH [BP+FF2E]
7C00:09D8 FFB62CFF        PUSH [BP+FF2C]
7C00:09DC 2BC0            SUB AX,AX
7C00:09DE 50              PUSH AX
7C00:09DF 9A2801FC44      CALL 44FC:0128
7C00:09E4 89868EFE        MOV [BP+FE8E],AX
7C00:09E8 899690FE        MOV [BP+FE90],DX
7C00:09EC C78672FE0000    MOV WORD PTR [BP+FE72],0000
7C00:09F2 EB04            JMP 09F8
7C00:09F4 FF8672FE        INC WORD PTR [BP+FE72]
7C00:09F8 83BE72FE05      CMP WORD PTR [BP+FE72],+05
7C00:09FD 7C03            JL 0A02
7C00:09FF E94201          JMP 0B44
7C00:0A02 52              PUSH DX
7C00:0A03 50              PUSH AX
7C00:0A04 9A4602FC44      CALL 44FC:0246
7C00:0A09 8946EC          MOV [BP-14],AX
7C00:0A0C 8956EE          MOV [BP-12],DX
7C00:0A0F FFB690FE        PUSH [BP+FE90]
7C00:0A13 FFB68EFE        PUSH [BP+FE8E]
7C00:0A17 9AF201FC44      CALL 44FC:01F2
7C00:0A1C 8946F0          MOV [BP-10],AX
7C00:0A1F 8D46E6          LEA AX,[BP-1A]
7C00:0A22 16              PUSH SS
7C00:0A23 50              PUSH AX
7C00:0A24 9A8202C93C      CALL 3CC9:0282
7C00:0A29 8346E80A        ADD WORD PTR [BP-18],+0A
7C00:0A2D FFB690FE        PUSH [BP+FE90]
7C00:0A31 FFB68EFE        PUSH [BP+FE8E]
7C00:0A35 B80100          MOV AX,0001
```

```
7C00:0A38 50            PUSH AX
7C00:0A39 9A7E01FC44    CALL 44FC:017E
7C00:0A3E 89868EFE      MOV [BP+FE8E],AX
7C00:0A42 899690FE      MOV [BP+FE90],DX
7C00:0A46 EBAC          JMP 09F4
7C00:0A48 B084          MOV AL,84
7C00:0A4A 50            PUSH AX
7C00:0A4B 9A1003443D    CALL 3D44:0310
7C00:0A50 C746E88C00    MOV WORD PTR [BP-18],008C
7C00:0A55 FFB62AFF      PUSH [BP+FF2A]
7C00:0A59 FFB628FF      PUSH [BP+FF28]
7C00:0A5D B85C00        MOV AX,005C
7C00:0A60 50            PUSH AX
7C00:0A61 9A2801FC44    CALL 44FC:0128
7C00:0A66 89868EFE      MOV [BP+FE8E],AX
7C00:0A6A 899690FE      MOV [BP+FE90],DX
7C00:0A6E 52            PUSH DX
7C00:0A6F 50            PUSH AX
7C00:0A70 9A4602FC44    CALL 44FC:0246
7C00:0A75 8946EC        MOV [BP-14],AX
7C00:0A78 8956EE        MOV [BP-12],DX
7C00:0A7B FFB690FE      PUSH [BP+FE90]
7C00:0A7F FFB68EFE      PUSH [BP+FE8E]
7C00:0A83 9AF201FC44    CALL 44FC:01F2
7C00:0A88 8946F0        MOV [BP-10],AX
7C00:0A8B 8D46E6        LEA AX,[BP-1A]
7C00:0A8E 16            PUSH SS
7C00:0A8F 50            PUSH AX
7C00:0A90 9A8202C93C    CALL 3CC9:0282
7C00:0A95 2AC0          SUB AL,AL
7C00:0A97 50            PUSH AX
7C00:0A98 9A1003443D    CALL 3D44:0310
7C00:0A9D 8346E80B      ADD WORD PTR [BP-18],+0B
7C00:0AA1 FFB690FE      PUSH [BP+FE90]
7C00:0AA5 FFB68EFE      PUSH [BP+FE8E]
7C00:0AA9 B80100        MOV AX,0001
7C00:0AAC 50            PUSH AX
7C00:0AAD 9A7E01FC44    CALL 44FC:017E
7C00:0AB2 89868EFE      MOV [BP+FE8E],AX
7C00:0AB6 899690FE      MOV [BP+FE90],DX
7C00:0ABA 52            PUSH DX
7C00:0ABB 50            PUSH AX
7C00:0ABC 9A4602FC44    CALL 44FC:0246
7C00:0AC1 8946EC        MOV [BP-14],AX
7C00:0AC4 8956EE        MOV [BP-12],DX
7C00:0AC7 FFB690FE      PUSH [BP+FE90]
7C00:0ACB FFB68EFE      PUSH [BP+FE8E]
7C00:0ACF 9AF201FC44    CALL 44FC:01F2
7C00:0AD4 8946F0        MOV [BP-10],AX
7C00:0AD7 8D46E6        LEA AX,[BP-1A]
7C00:0ADA 16            PUSH SS
7C00:0ADB 50            PUSH AX


; Lot's of code Huh?
```

```
7C00:0ADC 9A8202C93C      CALL 3CC9:0282
7C00:0AE1 C746E8BC00      MOV WORD PTR [BP-18],00BC
7C00:0AE6 FFB690FE        PUSH [BP+FE90]
7C00:0AEA FFB68EFE        PUSH [BP+FE8E]
7C00:0AEE B80100          MOV AX,0001
7C00:0AF1 50              PUSH AX
7C00:0AF2 9A7E01FC44      CALL 44FC:017E
7C00:0AF7 89868EFE        MOV [BP+FE8E],AX
7C00:0AFB 899690FE        MOV [BP+FE90],DX
7C00:0AFF 52              PUSH DX
7C00:0B00 50              PUSH AX
7C00:0B01 9A4602FC44      CALL 44FC:0246
7C00:0B06 8946EC          MOV [BP-14],AX
7C00:0B09 8956EE          MOV [BP-12],DX
7C00:0B0C FFB690FE        PUSH [BP+FE90]
7C00:0B10 FFB68EFE        PUSH [BP+FE8E]
7C00:0B14 9AF201FC44      CALL 44FC:01F2
7C00:0B19 8946F0          MOV [BP-10],AX
7C00:0B1C 8D46E6          LEA AX,[BP-1A]
7C00:0B1F 16              PUSH SS
7C00:0B20 50              PUSH AX
7C00:0B21 9A8202C93C      CALL 3CC9:0282
7C00:0B26 B80100          MOV AX,0001
7C00:0B29 50              PUSH AX
7C00:0B2A 9AF4019324      CALL 2493:01F4
7C00:0B2F 83C402          ADD SP,+02
7C00:0B32 B047            MOV AL,47
7C00:0B34 50              PUSH AX
7C00:0B35 9A1003443D      CALL 3D44:0310
7C00:0B3A 8D46F6          LEA AX,[BP-0A]
7C00:0B3D 16              PUSH SS
7C00:0B3E 50              PUSH AX
7C00:0B3F 9A6A00843D      CALL 3D84:006A
7C00:0B44 83BEF2FE00      CMP WORD PTR [BP+FEF2],+00
7C00:0B49 7508            JNZ 0B53
7C00:0B4B FF4E84          DEC WORD PTR [BP-7C]
7C00:0B4E 7403            JZ 0B53
7C00:0B50 E9A7F9          JMP 04FA
7C00:0B53 FF76F4          PUSH [BP-0C]
7C00:0B56 8D867AFF        LEA AX,[BP+FF7A]
7C00:0B5A 50              PUSH AX
7C00:0B5B FFB62EFF        PUSH [BP+FF2E]
7C00:0B5F FFB62CFF        PUSH [BP+FF2C]
7C00:0B63 FFB62AFF        PUSH [BP+FF2A]
7C00:0B67 FFB628FF        PUSH [BP+FF28]
7C00:0B6B E88EF5          CALL 00FC
7C00:0B6E 8B86F2FE        MOV AX,[BP+FEF2]
7C00:0B72 5E              POP SI
7C00:0B73 5F              POP DI

; Here is the exit code I was talking about

7C00:0B74 8BE5            MOV SP,BP
7C00:0B76 5D              POP BP
7C00:0B77 CB              RETF
```

```
7C00:0B78 B85A06            MOV AX,065A
7C00:0B7B CB                RETF
7C00:0B7C B89006            MOV AX,0690
7C00:0B7F CB                RETF
```

Ok, after looking through all of that, can you tell me where to put the patch. Simple. How about right at the begining of the doc check right after the music routines (ie address 7C00:04B6). Hey yeah ... good idea. But how do we want to patch it. Well, since this is a higher level language, we just can't use RETF. We must reset the stack.

Since I hate large patches, a simply decided on the follow patch

```
7C00:04B6 E9BB06            JMP B74
```

Ok, by jumping to 0B74, we still get the music but the actual doc check is not executed. But there is still a problem. Remember how I said that AX was tested after the doc check. Well, we still have to fake the check. The easiest way, is to simply NOP the condition jmp. Here is the section of code again

```
45E2:0235 9A46010F4A      CALL 7C00:146     ; Call to Doc Check
45E2:023A 83C404          ADD   SP,+04
45E2:023D 0BC0            OR    AX,AX
45E2:023F 7465            JZ    02A6
```

If you remember, when you enter the right code, AX will be set to 0001 when we exit to 45E2:023A. If we OR 0001 and 0001 we get 0001. Here is the binary ...

```
                    0000 0000 0000 0001   ( remember OR means
                                            if  either is bit
                or  0000 0000 0000 0001       is 1 )
                    ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ
                    0000 0000 0000 0001
```

Clearly we don't want to branch at the JZ at 45E2:023F. So, to finish the patch we simply NOP that jmp.

Oh boy.. that was hard. So let's test it out. But first, a little forsight. We will need a unique string of bytes to search for when making the patch. I say we use the code from 7C00:04C4 to 7C00:04CE and from 45E2:0235 to 45E2:023F. Yea, write down the hex equivelent and then restart. Again break in right after the switch to graphics. Now add the patch (ie A 7C00:04B6 <ENTER>, etc.). Now execute the program.

SHIT! It worked, we are fucking amazing. Ok, now adding the patch permenatly. Using PCTOOLS (or whatever) search the file STARCON.EXE for the bytes mention above (ie: C746F60B00C746F87900C746FA2801) But wait, now matches...Hmmm strange. It was there just a minute ago...but wait there... another file STARCON.OVL (as we all know .OVL mean OVERLAY). Let's try searching this one.

There we go, that's better (it should should up on the 13 sector read in). Now to add the patch. Simply find the search bytes and the go backwards until the first occurance of the hex byte 9A. Add the patch here. Save it.

Next, add the patch to 45E2:023F. Search for the bytes 83C4040BC07465. The should appear on sector 3 (give or take a few sectors). Now simply change the 2 bytes 74 65 to 90 90 and save the sector. Now, you are good to go.

Well shit, this has been some hell of a textfile. 1113 lines in all. But what detail. Ok I hope you learned something from all of this. And this end the first part of CRACKING 101 - the 1990 edition. From here out all lessons (lesson 5 and up) will be released on their own.

I would like the thank Phantom Phlegm for pushing me to finish this shit.

Till lesson 5 this is Buckaroo Banzai, signing off.

OH... I can be reached for personal help via E-MAIL on LORD WOLFEN's CASTLE or TOS...