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 decided 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 "Retirement" 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 by-pass 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 com-

plete 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.

10

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).

11

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.

Chapter II Example Cracks

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, continues 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 RENname 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
```

13

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 hexa-decimal 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.

20

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 reloaded 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 pro-
gram.

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

Then insert the copy disk in the A drive and invoke DOS DE-
BUG.

```
>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.

23

```
-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 re-
turned 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 either 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.

25

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

26

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 ourselves.

```
          -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.