About the program

Welcome to Hackman the power tool for debugging and editing source code. Its purpose is to enable you to edit, modify or monitor the source code of your applications.

You may use this program as long as you don’t infringe any copyright laws. You do not have the right to change the source code of an application if you haven’t got the permission of its author(s).

Introduction to Assembly

If you are already well familiar with the assembly language, you may wish to skip this section.

Why assembly? Debugging demands the knowledge of the assembly language. If you wish to become a serious programmer, you might like to read up more about this fascinating language. This section will only give you enough info for intermediate level of programming with assembly.

NOTE: This version of Hackman does not allow you to assemble directly parts of code. To do this, you may use Dos Debug that comes with Dos version 6.22 or Windows 95.

Registers

One of the neato things that you will be fooling around most often with are called the registers. Registers are like variables (such as in BASIC) that are located within the CPU itself. These registers may hold a positive integer from 0 to 255 or from 0 to 65535. They can also hold negative integers from -128 to 127 or from -32768 to 32767. The registers are given names as follows:

- *AX* => *accumulator* - this register is most commonly used for mathematical or I/O operations
- *BX* => *base* - this register is used commonly as a base or a pointer register (we'll talk more about this later)
- *CX* => *count* - used commonly for counting instructions such as loops
- *DX* => *displacement* - much like the base register

The registers stated above are considered general purpose registers, since they can basically be used to store whatever the user wants.

Segment registers

The following are called the **segment** registers:

- *CS => code segment* - the block of memory where the code (instructions are located)
- *DS => data segment* - the block of memory where data can be accessed. In block move operations in which huge blocks of memory are moved this is commonly the segment in which the CPU reads from.
- *ES => extra segment* - also another data segment. In block move operations in which huge blocks of memory are moved, this is commonly the segment in which the CPU writes to.
- *SS => stack segment* - this is the block of memory in which the CPU uses to store return addresses from subroutines.

Monitor registers

There are other registers that we use to see what the program is doing. These registers can also be change in debuging . Included are the following:

- *SI* => *source index* - this register is used in conjunction with block move instructions. This is a pointer within a segment (usually DS) that is read from by the CPU.
- *DI* => *destination index* - this register is also used in conjunction with block move instructions. This is a pointer within a segment (usually ES) that is written to by the CPU.
- *BP* => *base pointer* - a pointer used commonly with the stack segment
- *SP* => *stack pointer* - another pointer used commonly with the stack segment (this one, you don't touch)

Using registers

Putting some number in these registers is quite simple once you realize how they work. Assuming you are using debug, just type in "R {enter}".

You should see a bunch of info, of which are four of the above mentioned registers. Now, type in "RAX {enter}". Then type in a number like 8FABh. Type in "R" again and noticed how the accumulator (AX) has change its number.

These general purpose registers can also be "split" in half into its higher and lower order components. Instead of having one register AX, you can have two registers, AH and AL. Note however that while you have a range of 0 to FFFFh for AX, you will now have a range of 0 to FF for AH and AL. You cannot change these directly in debug, but be aware that programs will use it. If AX contains 0A4Ch, then AH will contain 0Ah and AL will contain 4Ch.

Flag registers

One last important register is the **flags** register. These registers control how certain instruction work, such as the conditional jumps (in BASIC, they are like IF-THEN's). They are stored as bits (0's or 1's) in the flags register. We will most often use: zero => ZR/NZ (zero/not zero) - tells you whether an instruction (such as subtraction):

- yielded a zero as an answer sign => NG/PL (negative/positive), tells you whether aninstruction
- yielded a positive or negative number carry => CY/NC (carry/no carry) - tells you whether an instruction needed to carry a bit (like in addition, you carry a number over to the next digit). Various system (BIOS) functions use this flag to denote an error.
- direction => DN/UP (decrement/increment) - tells a block instruction to either move forward or backwards in reads and writes

MOV Command

MOV {destination},{source}.

Assuming that you are using debug type in "A {enter}". You will see a bunch of number to the left. You can think of these as line numbers. Now type in "MOV AX,7A7A {enter}". Then type "MOV DX,AX" and so on until your program looks similar to the one below: (type "U 100" to see)

```
xxxx:0100 B8A77A MOV AX,7AA7
xxxx:0103 89C2 MOV DX,AX
xxxx:0105 B90000 MOV CX,0000
xxxx:0108 88D1 MOV CL,DL
xxxx:010A 890E0005MOV [0500],CX
xxxx:010E 8B160005MOV DX,[0500]
xxxx:0112 BB0200 MOV BX,0002
xxxx:0115 26A30005MOV ES:[0500],AX
```

Press enter again until you see the "-" prompt again. You are ready to run your first program. Type "R {enter}" and note the values of the general purpose registers. Then type in "T {enter}". Debug will automatically display the registers after the execution of the instruction.

What is in the AX register? It should be 7AA7h. Now, "T" again. What is in the DX register? It should also be 7AA7h. Trace again using "T" and note that CX should be 0 if it was not already. Trace again and note what is in the CX register. It should be 00A7h. Now trace another step.

Purpose: It is moving the contents of CX into memory location 500h in the data segment (DS). Dump the memory by typing in "D 500". The first two two-digit numbers should be the same as in the CX register. But wait a minute you say. They are not the same. They are backwards. Instead of 00A7h, it is A700h. This is important. The CPU stores 16 bit numbers in memory backwards to allow for faster access. For 8 bit numbers, it is the same.

Now, continue tracing. This instruction is moving the memory contents of address 500h into the DX register. DX should be 00A7h, the same as CX regardless of how it looked in memory. The next trace should be nothing new. The next trace again moves the contents of a register into memory. But notice it is using the BX register as a displacement. That means it adds the contents of BX and 500h to get the address, which turns out to be 502h. But also not the "ES:" in front of the address. This additional statement tells the CPU to use the extra segment (ES) rather than the data segment (DS which is the default). Now dump address 502h by entering "D ES:502" and you should see A77Ah, which is backwards from 7AA7h.

CMP - J? Command

Another instruction you will see quite often is the CMP or **compare** instruction. This instruction compares the two "variables" and changes the flags register accordingly. The source and destination operands are the same as those for the move instruction. Let's consider an example in which the AX register holds 21 and the BX register holds 22. Then "CMP AX,BX" is performed.

The compare instruction is like a subtraction instruction, but it doesn't change the contents of the AX register. So, when 22 is subtracted from 21, the answer will be -1, but we will never see the answer, only the flags which have resulted from the operation. Number 21 is less than 22, so the carry flag and the sign flag should be set. Just remember that when the carry flag is set, the first number is less than the second number. The same is true for the sign flag. Why have two flags if they tell us the same thing? This is more complicated and you should not concern yourself with it. It requires knowledge of hexadecimal arithmetic, the denotation of signed and unsigned integers.

After the compare instruction, there will most likely be a conditional jump instruction after. If we wanted to jump if AX is less than BX (which it is), then there would be an instruction like "JB 200". This instruction says Jump if Below to instruction 200h. What about if we wanted to jump if AX is greater than BX. Then we might have "JA 200". This is read Jump if Above to instruction 200. What about AX equal to BX. We would then have "JZ 200" or "JE 200". (Please note that the previous instructions are synonymous.) This is read Jump if Equal to instruction 200h. Here are the jumps you will most likely encounter:

i) JB/JNAE CF=1 Jump if below/not above or equal (unsigned)
ii) JAE/JNB CF=0 Jump if above or equal/not above (unsigned)
iii) JBE/JNA CF=1 or ZF=1 Jump if below or equal/not above (unsigned)
iv) JE/JZ ZF=1 Jump if equal/zero
v) JNE/JNZ ZF=0 Jump if not equal/not zero
vi) JL/JNGE SF not equal Jump if less/not greater or to OF equal (signed)
vii) JGE/JNL SF=OF Jump if greater or equal/not less (signed)
viii) JLE/JNG ZF=1 or SFJump is less or equal/not not equal OF greater (signed)
ix) JG/JNLE ZF=0 or SF=OF Jump if greater/not less or equal (signed)
x) JS SF=1 Jump if sign JNSSF=0 Jump if no sign
xi) JC CF=1 Jump if carry JNCCF=0 Jump if no carry
xii) JO OF=1 Jump if overflow JNOOF=0 Jump if not overflow
xiii) JP/JPE PF=1 Jump if parity/parity even
xiv) JNP/JPO PF=0 Jump if no parity/parity odd

JMP Command

This instruction does what it suggests. It jumps too different sections of code. Several forms of the jump instruction include:

```
xxxx:0208 EBF6 JMP 0200
xxxx:020A 3EFF24 JMP DWORD PTR DS:[SI]
```

The first instruction jumps to an address within the segment. The latter instruction jumps to an address pointed to by DS:SI. The DWORD says that this will be a far jump, a jump to a different segment. So, if the double word that is pointed to by DS:SI contains 1000:0040h, then, the instruction will jump to 1000:0040h whereas the previous jump instruction will jump within the current segment.

CALL Command

This instruction calls another procedure and upon it's completion, will return to calling address. For example, consider the following block of code:

```
xxxx:1002 E8BB46 CALL 56C0
xxxx:1005 7209 JB 1010
xxxx:1007 0C00 OR AL,00
```

The first line calls another procedure at "line number" 56C0h. Upon its completion, the instruction pointer will point to the second line. Note that there is a "JB” instruction. Programs often use the carry flag to signal errors. If the call instruction called a check procedure instruction to see if you have entered a wrong code or something, it may return with the carry flag set. The next instruction would then jump if there was an error to an exiting procedure. Note, this is a near call. A program can also have far calls just like jumps.

INT Command

This instruction is much like the call (generate) instruction. It also transfers control to another procedure. However, the number after the INT instruction does not point to an address. Instead, it is a number pointing to an address that is located in something called an interrupt vector. You will commonly see and use "INT 10", "INT 21", "INT 13". Just know that they are something like calls to procedures.

LODS? - STOS? Commands

These instructions either load in or store a byte or a word to or from memory. The DS:SI register pair points to the source data. These are the registers the CPU will use when reading from memory using the LODS instruction. The AX/AL register will hold the number to either read from or write to the memory. So, if DS:SI points to a byte which is maybe 60, then a "LODSB" instruction will load in the number 60 into the AL register.

A LODSB or STOSB will use the AL register while the LODSW or STOSW will use the AX register. The STOS writes whatever is in the AX/AL register to the memory pointed to by ES:DI. If ES:DI points to 100:102h and if AL held 50, then the byte at 100:102h will hold 50. After the instruction is finished, the CPU will either increment or decrement SI or DI according to the status of the direction flag. If SI was 100h and a "LODSW" instruction was performed with a cleared direction flag (forward), the SI will now point to 102h.

MOVS? Command

This instruction gets a byte or a word from the data pointed to by DS:SI and copies it to the data pointed to by the ES:DI address. When the instruction is finished, SI and DI will be incremented or decremented accordingly with the status of the direction flag. So, if DS:SI pointed to a byte with the number 30, a "MOVSB" instruction would copy into the byte pointed to by ES:DI the number 30.

REP Command

The REP instruction in front of a MOVS/LODS/STOS would cause the MOVS/LODS/STOS instruction to be repeated for a number of times specified in the CX register. So, if CX contained 5, then "REP STOSB" would store whatever was in the AL register into the byte pointed to by ES:DI five times, increasing DI each time.

LOOP Command

The LOOP instruction repeats a block of instructions for a certain number of times. This number will be held in the CX register. Each time we reach this instruction, the CPU will decrement the CX register and jump to a specified instruction until CX becomes zero. This instruction looks like "LOOP 1A00" where the number indicates the instruction address to loop to.

Arithmetic operators

Arithmetic instructions allow you to perform various arithmetic function of data. "ADD" and "SUB" work the same way as "MOV" instructions do in that it subtracts whatever is in the source register from the destination register and stores it in the destination register. The "MUL" and "DIV" instructions are a bit more complicated and they are not used as intensively as the "ADD" or "SUB" since they are slow.

Disk protection

There are basically two different forms of copy protection that you could create: the disk based and manual based copy protection schemes.

With **disk based** schemes, software often reads from specific sectors on a disk to determine the disk's validity. When you perform a disk format, the disk is formatted with specific sector sizes. Once the sector size changes, DOS cannot recognize it, thinking that it is a bad sector. Since this looks like a bad sector, a simple DISKCOPY will not work in copying such disks. Interrupt 13h (the assembly mnemonic is INT 13) was commonly used to handle such copy protections. It is now very rare to encounter the once famed INT 13h copy protection method nowadays since it was quite easy to defeat. Any professional commercial software (including yours) will often use their own custom based disk I/O routines. This involves intimate access to I/O ports using IN and OUT instructions. It is probable that the I/O functions might be called from a "CALL" instruction.

Another disk based scheme used to denote legality of software is used during the installation process of the software. With certain programs, when you install it, it copies the files into the hard drive. But it also sets a specific sector in the hard drive so that the program can recognize it. This disk-based copy protection is almost completely out of the software industry.

However, a more difficult copy protection scheme has arisen that may sometimes prove to be even more difficult to crack. These schemes are commonly known as the **doc checks** in which the user must have a copy of the manual to bypass the protection. With programs compiled as true assembly (you can call them "normal" programs), these protections are not too bad to trace through. With programs that run scripts, this can he a real chore however. It is like running a program within a program.

As if these copy protection schemes weren't enough, software companies have also added trace inhibition schemes to their code. This means that it is much time consuming for someone to try and trace through your code. On the other hand if someone knows how these things work, it should not be too much of a problem for him. Run-time compression / decompression and encryption / decryption of files also make changes to the program difficult. In this case, the loader sure comes in handy. Also, when the data within the file changes due to overlays, loaders are also good to use.

Working with disk protection

I have previously mentioned that INT 13h copy protection schemes are hardly ever used anymore. Nevertheless, it would be good practice for someone to learn how to create and understand the code. You will most likely see INT 13h used with function 2, read sector. This means that:

- *AH* => will contain the number 2 (function 2)
- *AL* => the number of sectors to read in. This is commonly only 1 since you just want to check a few sectors for disk validity.
- *CH* => will contain the cylinder number
- *CL* => will contain the sector number
- *DH* => will contain the head number
- *DL* => will contain the drive number 00h - 7Fh for floppies 80h - FFh for fixed disks
- *ES:BX* => will point to the address into which the data read from the disk will be written to.

Also refer to:
Advanced discussion

Advanced discussion

Upon the return for interrupt 13h, if the carry flag is set, that means that the program could not read the sector, and therefore the disk is valid. If the carry flag is clear, that meant that INT 13h could read the sector properly and so the disk would be bad in the eyes of the program, thinking it was a copied disk.

Before creating your disk protection system, we should be aware of the difference between debug's "T" and "P". "T" is the trace instruction, which tells it to follow instructions step by step. That also means that in LOOP or REP instruction, the trace will patiently go through the loop until finished. Also, during CALL instructions, trace will go into the call and execute the instructions pointed to by the call instruction. The "P" command is similar to the "T" but with the difference in that it traces over instructions. That means that if it use a LOOP or REP, it will quickly finish up the loop and point to the next instruction. With a CALL, the "P" (proceed) will not go into the subroutine. Instead, it will just execute the procedure, then point to the next instruction.

But where should you place your protection within an executable? Run your careful note of when things happen. You should notice (if you have used) an intro screen, then the music pops up, then the menu comes out. Decide with extreme caution where to put your protection. Use the most peculiar part of your application to do this.

Notice this so you will know where you are in your program using the unassembler. Once you have done that, you can begin debugging your program. When you are just about to execute the step, try to remember the segment and offset of the instruction. The segment is the number to the left of the colon while the offset is the number to the right. There are basically two different methods to engage your protection system:

Method 1: Exit from copy protected CALL

You should decide how your code would access the drive A: for example. Assuming that it is done with a CALL instruction you should be aware that someone may try skipping the CALL instruction.

It is not tough to accomplish this using even Debug. For example type in "RIP {enter}". Then type in the address of the next instruction. Then execute the do or die instruction, "G". If you fail, you could try using the "T" command once and start using "P" again.

Method 2: Return from copy protected CALL

There is another way to create a disk protection (or better to call your disk protection). To use an instruction that causes the program to jump because of a carry flag. This could be acieved through a JMP command.

Is it safe? No. You could easily fool it around with this carry. INT 13h copy protections are usually simple enough for you to just change the carry flag to allow the program to bypass the copy protection.

Conclusion

So it is not wise to use disk protection anymore. People with a minimum of assembly knowledge could easily mess around with your protection and have your program cracked. But it still remains a good idea for small programs that you just don't want **everybody** to copy them freely.

Files menu

Files menu consists of the following functions:

Open file: opens a file to the editor. Any previous files opened to the editor will be closed without saving changes or prompting to.

Change offset: you may declare the start point of the main source of a file. This option ensures that you are able to skip headers or other garbage info.

Anatomy: gives you general information about the selected file (attributes, length, location) and some code information (type, offset).

Properties: allows you to change file's properties. Setting the file as Read-Only does not allow you to save changes.

Backup: creates a quick backup of the file in case you wish to undo your changes.

Copy file: same as backup, but you can control the destination file by entering its name.

Delete file: you may delete a file regardless if it is Read Only or not.

Exit: exits the program without saving changes or prompting you to save them.

Editor menu

The editor menu consists of the following functions:

Edit: gives the focus to the editor preparing you to edit the source of the selected file. It's exactly the same as clicking the first hex byte of the editor monitor.

Find: finds a sole or a sequence of hex or ascii bytes within the source code of the selected file.

Find next: repeats last find command.

Search mode: you may select Hex or Ascii search mode if you want to search using a Hex or an Ascii key respectively.

Replace mode: you may select Hex or Ascii replace mode if you want to replace a Hex or an Ascii byte sequence respectively.

Watches…: access the watch manager. You may put up to 20 expression to trace. Many customization options as per request.

Show (or hide) watches: enables (or disables) watches.

Read menu

Read menu has the following sub-commands:

Current: reads again current page in order to undo changes that you haven't saved yet.
Next: reads next page. Equivalent to Page Down.
Previous: reads previous page. Equivalent to Page Up.
Jump to page: goes to page x, as defined by you. Note that if the selected file consists of one page only, you may not use the jump to command.
Jump to offset: goes to offset x, as defined by you regardless the current page.

Write menu

Has only the **write** command. This was done on purpose, to assure that you don't accidentally press write as you edit a source code. You may press F5 alternatively.

Note: you can't undo changes unless you have a backup file of the one that you edit.

Configuration menu

You have access to the following functions:

Conversion modes: convert source code to:

i) Hex & Ascii (the only editable mode)
ii) Ascii only
iii) Hex only
iv) Binary
v) Octal

Lettering mode: applicable to Hex & Ascii or Ascii only modes.

Standard set: uses only the standard 128-character set

Graphic: uses all available graphic characters.

Tuning: applicable to Hex & Ascii only mode.

Normal: represents Ascii keycodes using default values.

Single parameter: uses a single parameter to decrypt the source code. You should specify the single parameter through the Optimization… dialog box.

Complex algorithm: uses an algorithm specified by you in order to decrypt the source code. You should load an algorithm before this, through the Optimization… dialog box.

Monitors setup: control the output monitors by defining the number of columns per page and the amount of possible blank space between successive columns.

Unassembler setup: you may force the disassembler engine to represent a part of a code as you want in order to trace it fast.

Printer setup: it's the control panel for all printings within Hackman.

General Outlook: a couple of useful properties like enable/disable splash screen and autostart behavior.

Optimization: use this dialog box to specify the single parameter or import an algorithm.

Tools menu

The tools menu includes:

Unassembler: returns the assembly code of any file, including those that actually weren't build to be executable (e.g. ASCII files).
Encrypt: encrypts any file using a password up to 255 characters.
Decrypt: decrypts any file encrypted with Hackman's encrypt method using the password provided during the encryption.
Calculator: a simple calculator for Hex numbering system.
PlugIns: unofficial documentation and/or programs provided for Hackman by third party programmers.

Help menu

The help menu consists of the following options:

Contents: The path to books online for Hackman.
License: what you should be aware of before using this program.
What's new: a brief report to new functions.
Known issues: a brief description of known bugs and issues about Hackman.
Memory resources: depicts the memory status of your system.
System check: performs a quick check of your system to see if it meets the minimum requirements to run Hackman.
Frequently asked questions: answers to your most common questions.
Hackman on the web: connect to Hackman's site on the web through Netscape.
About…: version and miscellaneous information.

About cryptography

This article is an introduction to **cryptography**. For further information and how to's please refer to other articles within this book.

Cryptography is a technique used for encrypting (in our point of view) files. A file could be anything; from a simple ASCII text to a complicated zipped file or an executable. There are numerous algorithms used to encrypt a file and they are practically unlimited methods to achieve this.

Hackman comes with an encryption algorithm with its undo: the **decryption** of the encrypted file.

Every encryption method requires a **password**. It's a unique for every file or bunch of files sequence of Ascii characters (case-sensitive for us) which is used by the algorithm to produce a new file that has nothing in common with the original except of a mathematical sequence that helps us to go from one side to the other.

Using Hackman's Encryption

To encrypt a file follow these steps:

1. Open the file to the editor. To achieve this use command Open from the Files menu.
2. Select encrypt from Tools menu.
3. Type in a destination name in the first box including path and extension.
4. Type a password (3 to 255 characters). Note that the password is case-sensitive which means that FotiS is treated differently from fOtis.
5. Press **Encrypt** to start the encryption.

Pressing **clear** will clear both text boxes. Press **cancel** if you want to close this panel without performing the encryption. To decrypt the file you should follow the decryption process.

Note: Do not forget the password or you won't be able to decrypt your files.

Using Hackman's Decryption

To decrypt a file encrypted with Hackman's encrypt tool please follow these steps:

1. From the tools menu select decrypt.
2. Type in a destination name in the first box including path and extension.
3. Type the password used to encrypt the file (3 to 255 characters). Note that the password is case-sensitive which means that FotiS is treated differently from fOtis.
4. Press **Decrypt** to start the decryption.
5. The output filename is the same as the input filename but ends in "out". For example if you type C:\Temp\test.exe as the input file you should get C:\Temp\test.exeout as the output file.

About document checks

Another protection issue that we should discuss is the art of “Document checking”. We are going to analyze to you all the professional methods of protecting a program through the documentation.
As you realize, this could be done during the setup or when the main executable file runs for the first time. You are asked to insert a couple of words from a particular page of the manual. If the word(s) or phrase are correctly inserted then the program continues otherwise it exits.
Next, you’ll learn how to add documentation checking in your programs by learning how others do it and what to take care of, if you don’t want to see your program cracked.

Note: To be able to follow up the incoming analysis, you should already be familiar with the basics of assembly and know how to manipulate this stuff using HackMan.

See the theory of document check
and advanced discussion of doc-checks .

The theory of document check

Is it safe? The answer is that no, it is not as safe as other protection methods. That is, because others can sneak easily into your source and halt the checking routine. How is this achieved? There are two methods (usually). First is to enter right before the doc check occurs or while the doc check occurs.

To locate the source of the code that represents a certain action, you should use a graphical debugger such as Numega's SoftIce (copyrighted program from Numega). Suppose that one break's out on top of the doc check, then it's easy to have a much better view of what is going on or else he'll have to do some extra work, figuring out how the routine works.

So you should prevent others from detecting the start point of your doc-check routine.

For a cracker, the object is to get to the very top level of the routine and either of the previously described methods should lead him there. Since breaking out before the doc check occurs, usually puts you on top of the protection routine, there isn't much work ahead. The object is to simply find the **exact** start of the check. To do this, it's enough to trace through the source code. Then just remove the exit part of the routine.

So, if you decide that a doc check routine is suitable for your product remember that you should try to confuse crackers as much as possible with garbage looping and meaningless functions.

See advanced discussion of doc-checks
and analyzing the type of the doc check.

Determining start point of the doc check

Here we'll discuss step-by-step the procedure followed by pros to eliminate doc-checking. Learning this, should teach you how to write **your** protection routine compact and uncrackable.

What you need to do is find the start of the actual doc check. Most programs are written with Top/Down design in mind. In other words, each function is a collection of other functions. For example, the doc check procedure may make calls to several other procedures as shown in the next N/S flowchart of a typical doc check. (figure 1)

No.	DOC CHECK
1	Clear screen
2	Display locator info
3	Get input
4	Clear screen
5	Check input

(figure 1)

As it is obvious, by calling the doc check procedure, you actually call 5 others (and in turn, the doc check may be a part of another procedure). Now, a way we can tell we are inside the doc check procedure is by waiting for an individual step to occur. For example if the program displays the locator info but returns to the debugger (SoftIce) instead of waiting for input you can assume that you are **in** the doc check routine.

The object now, is to simply get back to the top. To do this, just keep jumping over instructions until you reach a RET, RETF or IRET command. You can trace through this and you'll be at the next instruction after the call to the doc check routine. You should write down this address. If you have to break out in the input routine, you basically do the same thing; only this time you know that you are already in the doc check.

At this point, you 'll have to figure out the input routine. This is usually easier than it sounds and it's only becoming difficult when dealing with event-driven input like mouse input. Your best bet is to try to find a loop. Now, depending on how the program gets it's input, you will end up breaking out in two places. If the program uses a BIOS/DOS routine that sits and waits for input, you will break out inside these routines. If it uses a BIOS/DOS routine that does not wait for input such as INT 16h function 1, then you will usually be inside the actual program.

You can make a pretty good guess at where you are, depending on your current segment address. If you are in a low segment or your segment is F000h, then you are probably in DOS or BIOS. If not, you are in the program. What you need to do, is look for the program to loop back on itself. For example, if you drop out in to the BIOS keyboard routine, you should see code similar to figure 2.

```
F000:E853 STI
F000:E854 HLT
F000:E855 JMP E857
F000:E857 JMP E859
F000:E859 JMP E85B
F000:E85B CLI
F000:E85C MOV AX,[SI]
F000:E85E CMP AX,[SI+2]
F000:E861 JZ E853
```

Figure 2

Let's analyze the code shown in fig 2. The 3 jumps (E855 to E859) are simply timing jumps. The last 3 instructions are the work-stuff. What they are doing is compare the head and tail length bytes for the keyboard buffer. If they are equal (i.e. no key was pressed) then the program re-executes the loop. This is exactly what we are looking for. Some sort of loop in the program. **Remember to avoid this type of code in order not to be traced easily by crackers.** If this part of code is traced then all that's left is executing the code (set a breakpoint at F000:E863). If we start jumping through code we could find a screen saying "You have entered the wrong/right code". Control is returned to the debugger and we can follow the instructions back to the top of the doc check routine by finding RET,RETF or IRET instructions.

See analyzing the type of the doc check and making a crack for the routine.

Analyzing the type of the doc check

There are (mostly) two different types of doc checks. Before cracking a program, we should check out what type of doc check is used. That's pretty simple. Once we've found the start of the doc check, we need to understand the theory behind the two types.

We shall call the first type **return by register** which is like a higher-level language function that returns a Boolean (true or false) value. Figure 1 is an example of this.

```
Function DocCheck : Boolean;
        BEGIN
                ..Code to do the check
                .. function returns true if
                .. check was ok.
        END;
        BEGIN
                IF DocCheck THEN
                        MainProgram
                ELSE
                        PirateExit;
        END.
```

figure 1

What happens here is that the doc check function returns a boolean true if the user has entered the correct word or phrase. All compilers use AX to return a boolean value from a function. This means that the check will be something that compares AX to some value. You may use Hackman's unassembler to determine the exact offset. Next figure 2, shows a typical type 1 doc check including call to check function and code to check the return value.

```
xxxx:0001 CALL DocCheck
xxxx:0003 OR AX,AX
xxxx:0005 JZ 0100
```

figure 2

DocCheck returns a value in AX register as shown in figure 1. The OR AX,AX statement tests to see if the value is 0. The JZ jumps if a value of 0 is returned. That simple.

Let's sneak in the second doc-check type. It is a routine that during the middle sets a value. Figure 3 is an example of this type written is pseudo-pascal code. Right at the end of the doc check routine, the program sets a global variable. This variable is checked later in the program. To tell if you have to deal with a type-2 doc check, if after the call to the routine the program just seems to go on with other business (i.e. no CMP AX,value, no OR AX,AX etc.) we can assume that it's type 2.

Note: sometimes you deal with both type 1 and 2.

```
VAR
        PirateCopy : Boolean;
PROCEDURE DocCheck;
        BEGIN
```

```
        ShowInfo;
        GetStr;

        If Str =Right THEN
                PirateCopy:=TRUE;
        ELSE
                PirateCopy:=FALSE;
END
BEGIN
        DocCheck
                …Does Something
        IF NOT PirateCopy
                MainProgram;
END.
```

figure 3

Our final step is to crack the routine.

Cracking the doc check routine

Now that we have found the start offset of the doc check and defined its type, it's time to do some real work. Our job is to set a breakpoint on the address of the call to the doc check.

Run the program until you reach at this address. For the time being, let's assume that we are dealing with type 1. You'll be looking at code like the fragment shown in figure 1. We want to figure out which branch of the jump at xxxx:0005 we need to take. To do this, simply execute the call to the doc check (do **not** trace in it, just put some false info) the program should make the check and exit back to the debugger. The jump will be made depending on the value of AX. Trace through the jump. If you take the branch and end up to xxxx:0100 then you should know that you **don't** want to take this branch cause you've entered wrong info. The opposite applies if you didn't take the branch.

To test this theory, rerun the program up to the point where computer checks the value (i.e. xxxx:0005). Now, reassemble the jump forcing (or not forcing depending on your theory) the program to take the right path.

```
xxxx:0001 CALL DocCheck
xxxx:0003 OR AX,AX
xxxx:0005 JZ 0100
```

figure 1

Use Hackman to write changes and you're ready! Suppose now that you have to deal with type 2. Find the offset where a call is being made to get the user input. Now you need to find where it sets the value. Enter a bogus input and start tracing the code. You should expect to encounter a fragment like the one shown in figure 1 or something like a conditional jump as shown in figure 2.

```
@@outerLoop:    LODSB
                SCASB
                LOOPE @@outerLoop
                OR CX,CX
                JNZ xxxx
```

figure 2

This code compares two strings and jumps if they are not equal. Note the OR CX,CX statement. It is checking to see if the end of the string has been met or if it exited because two characters did not match. If you had entered the wrong input, you would have taken the "JNZ xxxx". This takes you to the **bad** branch. Keep tracing until you come to a piece of code that changes a static memory location. This is where the doc check sets a global variable. Note the memory location **and** variable cause you'll need it later.

Eventually, you will see something like a move to the static memory location. Now that you have the address of the variable **and** the value that it should take, you must make a simple patch for the program. The simplest and easiest way to achieve this is to change the code at the beginning of the doc check to set the variable and then exit.

```
xxxx:0001 MOVE [xxxx],value
xxxx:0003 RET
```

It kill **completely** the doc check which means that you see nothing! You may use Hackman to accomplish this task. No doc check is un-crack-able and you should be aware of this. In fact, it's pretty easy to patch the checking routine.