kris kaspersky a.k.a nezumi
(translated by olga kokoreva)

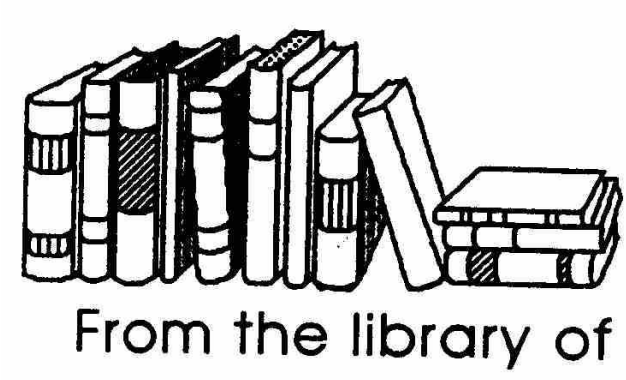
HACKER DISASSEMBLING UNCOVERED

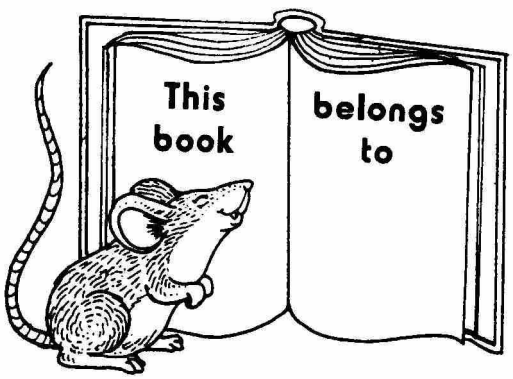
second edition

(draft version, chapter 10 only!)

鼠

visit **http://nezumi.org.ru**; **ftp://nezumi.org.ru**; send comments and remarks to **souriz at nezumi.org.ru**





annotation

what the disassembling is? temple of science? underground art? diabolic handicraft? black magic? or whole underworld hidden among machine commands or located elsewhere? how whole world could be fit between two assembling lines? it's not possible! but not for hackers, who easy discover unreal world. world without borders, without rules. very deep inside disassembling there is a wonderful world with off-the-wall rules. no any communication bridges there're around. only endless sands of the primeval forest haunted with sacred knowledge there is out there. the surviving becomes is the first hacker's a trick., a first hacker's trick you must to learn. without exact roadmap, without good guider and a couple buggy ghosts you'll be nothing, you'll be turned into dust and jammed! well, I give you everything you needs. I show you how to break into the code's hierograms and find out its their hidden meaning.

did you ever see the first edition of the book? that was a shit!!! I'd explained what any each assembler's brick does and which high-level construction its corresponds, but... I totally forgot to explain mention what you need have to do. you stay in front of the endless disassembling listing without any idea where to set afoot an inquiry. in essence, your guider was supposed to give you need more methodology. previous book definitely haves a lack of it, now the huge grasp has sealed. the book was has considerably rewritten, revisited and recast. keeping errata in mind, I'd fixed many errors, added quantity of the new chapters and updated the rests.: newer compiler and good old ones (like CGG and INTEL C/C++) now is described in details, I'd depicted 64-bit CPUs, LINUX/BSD disassembling specificity, far-out ulink linker, written by legendary Yury Haron, etcand many others stuff. in other words, I take give you *really* new book.

chapter 10: advanced debugging topics

In previous chapters of this part, the basic issues related to the use of disassembles and debuggers, and their use for code investigations was covered. This chapter, which concludes this part of the book, discusses advanced black hacking magic and prepares you for serious code investigations. This chapter acquaints you with various hacking tricks that allow more efficient hacking, including the following:

- Using SoftIce as a logger. There are lots of useful software tools intended for logging and tracking various system events (such as API spies). However, their capabilities are often limited. Therefore, hackers have no choice but create their own tools, however it takes time. Are there any tricks allowing for simplifying this task? Certainly there are. The use of SoftIce as logger is quick and dirty solution, but... another ones yet worse..
- Efficient techniques of working with breakpoints. Sometimes code diggers want to set a breakpoint on an arbitrary machine command (e.g., `jmp eax`). This topic is regularly discussed on forums and at newsgroups. Unfortunately, x86 processors do not provide such capabilities, however, I can be done in indirect way, describing bellow.
- Efficient techniques of quickly localizing protection mechanisms within large programs.

using SoftIce as a logger

Software tools that provide logging functionalities are numerous (consider API spies). Unfortunately, most of them don't satisfy the demands. They cause crash of "guinea-pig" program and die with empty log or are easily bypassed by viruses or protection mechanisms. As a result, the most important and valuable information about API functions used by the "guinea-pig" program were not included in the log. Furthermore, the size of generated logs usually impresses and horrifies. It is difficult to find useful information among millions of meaningless text strings. Many spies have some filters, but usually is very primitive, and helpless. In most cases it is possible to process the log by custom-external filter written in Perl or C. However, this approach is labor intensive. In addition, do not forget about situations in which you require information missing from the log (e.g., assume that you need to check whether a given API function is followed by the `TEST EAX,EAX` command.) Furthermore, what about situations in which you need to spy not only on API functions but also other events? For example, often it is necessary to sniff the protocol of data exchange between the "guinea-pig" application and some driver or even the hardware

SoftIce provides such possibilities. The suggested approach consists of creating a conditional breakpoint with intricate parameters and making the debugger output information into the log instead of popping up. Note that it is up to you to define the list of data items to include in the log, as well as the order of data output. The macros is a great thing, allowing the hacker to obtain a flexible and easily configurable logger that provides unlimited capabilities. The only problem remaining is to master these macros and use them to their greatest potential.

warming up

Before you start using SoftIce as a logger, it is necessary to configure it. Start **Symbol Loader** (Fig. 10.1), then choose the **Edit** | **SoftIce Initialization Settings** command from the main menu and increase the history buffer to several megabytes. The exact value depends on the specific task. The more information you need to collect during one session, the larger the buffer must be. Because Soft-Ice uses cyclic-buffer, it does not overflow after filling. Newer data simply overwrite the oldest records.

Go to the **Macros Definitions** tab (Fig. 10.2) and increase the number of simultaneously used macros from 32 (the default option) to any desired value (the allowed maximum is 256). Note that the latter modification is optional. For most tasks, 32 macros are enough.

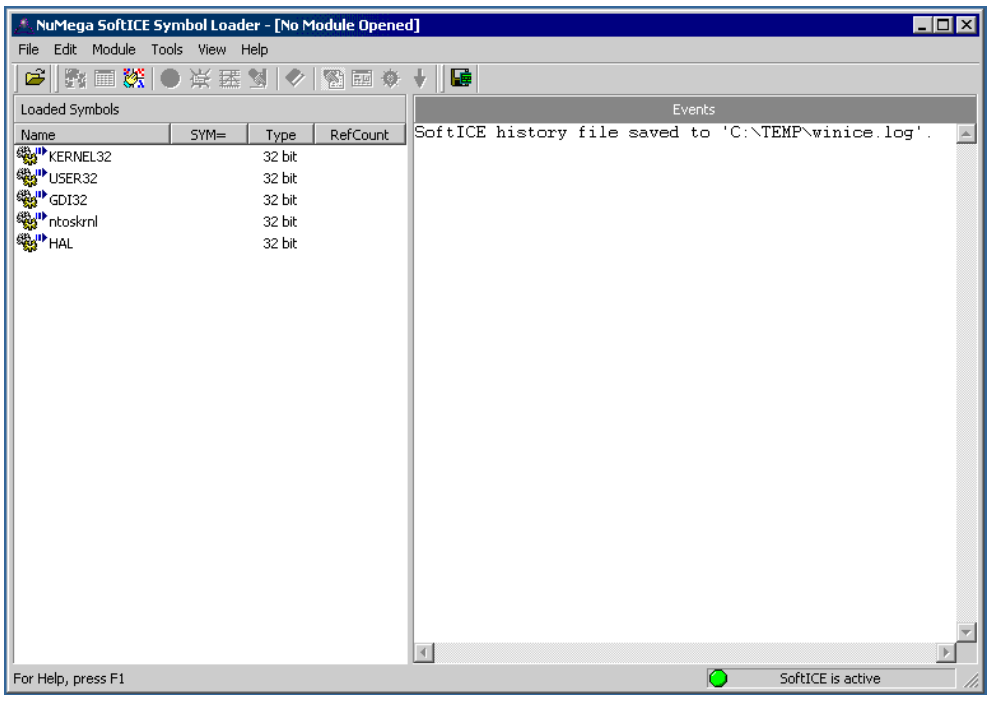


Fig. 10.1. NuMega SoftIce Symbol Loader

Well, lets try to trace the calls of `CreateFileA` function intended for opening devices and files. Create a custom breakpoint as follows: `'bpx CreateFileA DO "x;"'`. The `DO` keyword defines the sequence of the debugger commands that the debugger must execute after the breakpoint snaps into action. The commands must be separated by a semicolon. More details on the syntax rules can be found in the "Conditional Breakpoints" chapter of the SoftIce user manual. In the case being considered, only the `x` command is employed, which stands for immediate exit from the debugger.

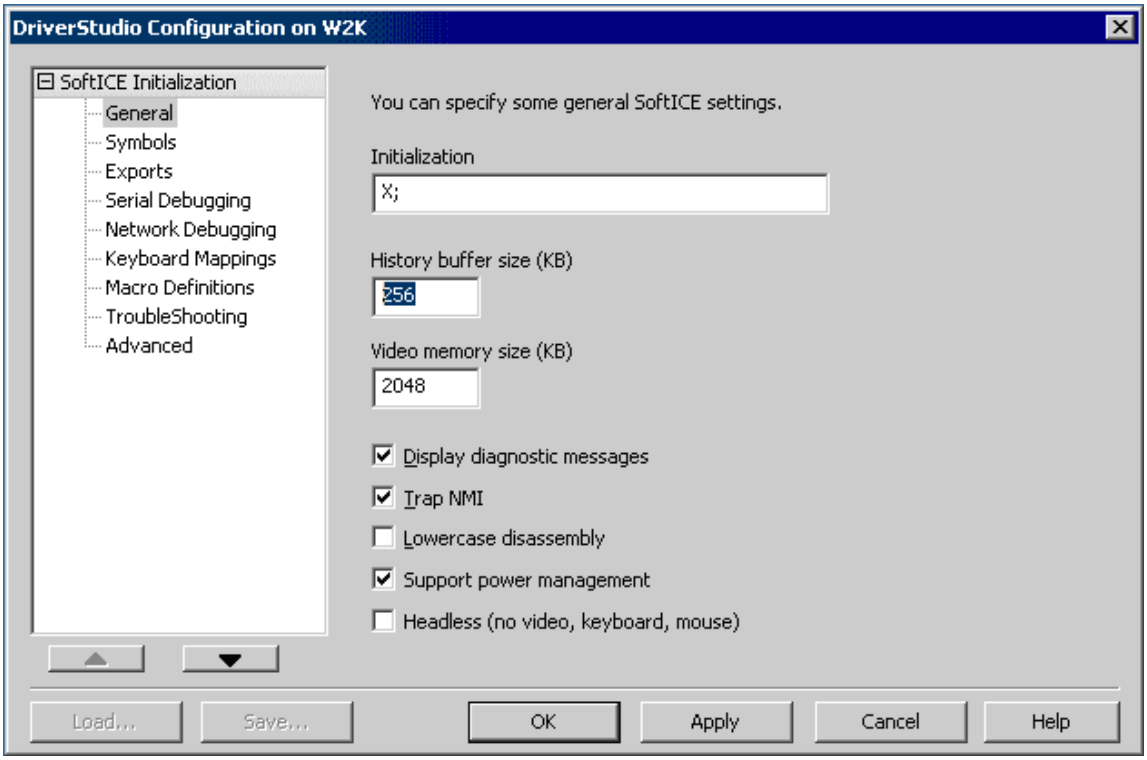


Fig. 10.2. Customizing the size of the SoftIce history buffer

Press <CTRL>+<D> to return into Windows and try to open some files. Having completed this, start Symbol Loader and save the SoftIce history in a file using the **File** | **Save SoftIce history as** commands. After a short interval, the new file will be created with the default file name of winice.log. The contents of this file will appear approximately as shown in Listing 10.1.

Listing 10.1. An example of the SoftIce log file contents

```
Break due to BPX KERNEL32!CreateFileA  DO "x;" (ET=1.44 seconds)
Break due to BPX KERNEL32!CreateFileA  DO "x;" (ET=940.19 milliseconds)
Break due to BPX KERNEL32!CreateFileA  DO "x;" (ET=14.51 seconds)
Break due to BPX KERNEL32!CreateFileA  DO "x;" (ET=19.23 milliseconds)
Break due to BPX KERNEL32!CreateFileA  DO "x;" (ET=13.88 milliseconds)
```

As you can see, the file contains lots of text strings, each describing the cause of the breakpoint actuation and when this event took place. At first glance, everything looks fine. However, there is no cause for joy. The information is practically useless, because it is unclear which file was opened at any given instance. Also, no information about success or failure of each operation is provided. Generally, the breakpoint requires considerable elaboration. An improved variant of the breakpoint is shown in Listing 10.2.

Note

SoftIce does not allow you to set two breakpoints to the same function. So, before creating a new breakpoint for a function for which there is already a breakpoint, it is necessary to delete the older breakpoint using the `bc n` *command, where "n" is number of the breakpoint*

Listing 10.2. A conditional breakpoint that outputs the names of all opened files

```
bpx CreateFileA DO "D esp->4 L 20; x;"
```

What has changed? Now the breakpoint will output names of opened files: `D esp->4 L 20`, where `D` is the command for displaying the dump, `esp->4` is the pointer to the first argument of the `CreateFileA` function (recall that this argument specifies the file or device to open), and `L 20` specifies the number of bytes for output (the value is chosen according to your requirements). Test the updated variant. To achieve this, press <CTRL>+<D>, exit the debugger, and start some program whose the actions must be traced. For example, let it be FAR Manager. Carry out some operations in the program being traced, then press <CTRL>+<D> again, enter the debugger, and issue the `bd 0` command to stop espionage. Exit SoftIce, start Symbol Loader, and save the history to the disk.

This time, the obtained result will appear approximately as shown in Listing 10.3.

Listing 10.3. An improved log displaying the names of files opened by the traced program

```
Break due to BPX KERNEL32!CreateFileA  DO "d esp->4 L 20;x;" (ET=3.64 seconds)
0010:004859E8 43 4F 4E 4F 55 54 24 00-43 4F 4E 49 4E 24 00 49  CONOUT$.CONIN$.I
0010:004859F8 6E 74 65 72 66 61 63 65-00 4D 6F 75 73 65 00 25  nterface.Mouse.%

Break due to BPX KERNEL32!CreateFileA  DO "d esp->4 L 20;x;" (ET=8.98 milliseconds)
0010:004859F0 43 4F 4E 49 4E 24 00 49-6E 74 65 72 66 61 63 65  CONIN$.Interface
0010:00485A00 00 4D 6F 75 73 65 00 25-63 00 25 30 32 64 3A 25  .Mouse.%c.%02d:%

Break due to BPX KERNEL32!CreateFileA  DO "d esp->4 L 20;x;" (ET=16.93 milliseconds)
0010:00492330 43 3A 5C 50 72 6F 67 72-61 6D 20 46 69 6C 65 73  C:\Program Files
0010:00492340 5C 46 61 72 5C 46 61 72-45 6E 67 2E 6C 6E 67 00  \Far\FarEng.lng.
```

Well, this is a different matter! Now the name of the file being opened is included in the log, which means that the improvised spy provides at least some useful information. However, this log still lacks such important data as the identifier of the process that has called an API function and the return code. Nothing can be easier than improving the newly created breakpoint (Listing 10.4).

Listing 10.4. The final version of a conditional breakpoint for logging purposes

```
bpx CreateFileA DO "? PID; D esp->4 L 20; P RET; ? EAX; x;"
```

Here the `? PID` command outputs the process identifier, the `P RET` command executes the API function being tracked and waits for the function to return a value, and the `? EAX` command returns the contents of the `EAX` register containing the API function's return code. The resulting log, produced by the breakpoint defined in Listing 10.4, is shown in Listing 10.5.

Listing 10.5. The final version of the log

```
Break due to BPX KERNEL32!CreateFileA  DO "? PID; D esp->4 L 20; P RET; ? EAX; x;"
000001DC  0000000476  "□▄"                                    ; PID
0010:0012138C 43 44 2E 73 6E 61 69 6C-2E 65 78 65 00 61 5F 65  CD.snail.exe.a_e
0010:0012139C 2E 65 78 65 00 00 00 00-00 00 00 00 00 00 00 00  .exe............
00000074  0000000116  "t"                                     ; Return code

Break due to BPX KERNEL32!CreateFileA  DO "? PID; D esp->4 L 20; P RET; ? EAX; x;"
000001DC  0000000476  "□▄"                                    ; PID
0010:0012138C 64 65 6D 6F 2E 63 72 6B-2E 65 78 65 00 61 5F 65  demo.crk.exe.a_e
0010:0012139C 2E 65 78 65 00 00 00 00-00 00 00 00 00 00 00 00  .exe............
00000074  0000000116  "t"                                     ; Return code

Break due to BPX KERNEL32!CreateFileA  DO "? PID; D esp->4 L 20; P RET; ? EAX; x;"
000001DC  0000000476  "□▄"                                    ; PID
0010:0012138C 64 65 6D 6F 2E 70 72 6F-74 65 63 74 65 64 2E 63  demo.protected.c
0010:0012139C 72 6B 2E 65 78 65 00 00-00 00 00 00 00 00 00 00  rk.exe..........
00000074  0000000116  "t"                                     ; Return code
```

This is worth studying. Thus, you made SoftIce provide the functions of a typical API spy. If desired, the filter can be elaborated further by adding new criteria. The log format also can be easily improved and complemented by new details. Note that you can output any details that you might require (e.g., the contents of the call stack).

You will encounter certain problems when implementing this approach. SoftIce pops up constantly and considerably reduces the performance. It is possible to make SoftIce log events without popping up. The debugger supports the `BPLOG` special function that always returns `TRUE` and keeps the debugger from popping up. Unfortunately, the `BPLOG` also suppresses the sequence of commands that follows the `DO` keyword, which makes creation of detailed logs impossible. Therefore, this method is not suitable for achieving the desired goal. Unfortunately, there are no other anti-pop-up tools available.

Garbage in the log is another source of headache. Useful data are mixed with other information output to the screen. Thus, it is impossible to speak about ease of use. Without writing a specialized report formatter, you will simply drown in tons of meaningless garbage. It is possible to write such a formatter using Perl. However, there is even easier approach. These are macros built into FAR Manager.

Start FAR Manager, press <F4>, and carefully view the log text (see Listing 10.5). As you can see, each fragment of the reported information starts with the `Break due to` string. This string will be used as a search string. Press <CTRL>+<.> to start macro recording, press <F7>, enter the search string (`Break due to`), and finally, press <ENTER>. Then press <SHIFT>+<→> to select the string fragment starting from the end of the `Break due to` string and ending at the start of the `CreateFileA` substring. This information is unnecessary, so press <CTRL>+<DEL> to cut it. Then again press <SHIFT>+<→> to go to the `DO` keyword. Select the fragment of unneeded text and delete it again. In other words, you can cut whatever you need from the text. Note that it is easier to do this than to describe it. After you complete macro creation, it is enough to apply it to the log the required number of times. To achieve this, assign a keyboard shortcut to the newly created macro and then press it the required number of times. The obtained result might appear approximately as shown in Listing 10.6.

Listing 10.6. The improved log cleaned up by the newly created macro

```
CreateFileA, PID:1DCh; NAME: CD.snail.exe;                RET: 74h
CreateFileA, PID:1DCh; NAME: demo.crk.exe;                RET: 74h
CreateFileA, PID:1DCh; NAME: demo.protected.crk.exe;      RET: 74h
```

The achieved result is good enough, especially if you recall that achieving it required only minutes. Macros are great tools that allow you to do nearly anything. Some purists might grin contemptuously, saying that this approach is amateurish, but this doesn't matter. The main goal was achieved within a surprisingly short amount of time, and this is the only thing that matters. If you want to use a more "professional" approach, then you will have to do a great deal of programming (e.g., in Perl).

more sophisticated filters

So far, only a standard logger similar to a typical API spy has been created. There are hundreds of such utilities, so was this task worth implementing? The answer is yes, because SoftIce is considerably less inclined to conflicts than most spies, especially when using hardware breakpoints like `bpm`. Thus, it successfully copes with the tasks that are unpractical when using other tools. This is especially true if you patch SoftIce first by installing the **IceExt** add-in module that hides the debugger from certain protection mechanisms.

Consider a slightly more sophisticated task than the one that was solved in the previous section. This time, the logger must spy on only files with names starting with the letter "a" instead of spying on all files. This problem can be solved as shown in Listing 10.7.

Listing 10.7. The breakpoint that logs access to files with names starting with the letter "a"

```
bpx CreateFileA if byte (*esp->4)=='a' DO xxx
```

The problem is that if the fully qualified file name is passed to the `CreateFileA` function, this breakpoint becomes impracticable because it checks only the first character of the file name. Unfortunately, the built-in toolset of SoftIce does not provide the substring search function. That's a pity. However, this minor drawback won't stop a hacker.To understand the suggested solution, note that, as a rule, the memory above the stack pointer is free and can be used according to the programmer's needs. What if you write a tiny assembly program, insert it there, and pass control to it? If successful, this will allow unlimited extension of the debugger's functionalities without using plug-ins, which usually are too bulky and poorly documented.

To execute a program in the stack, you will need an executable stack. Until now, this has not presented a problem. It was possible to execute any code in the stack without resorting to tricks. Recently, the situation changed dramatically. To thwart viruses and network worms, processor manufacturers, in cooperation with Microsoft, have introduced the protected stack. In the last versions of Windows XP, Windows Server 2003, Windows Vista, and Windows Server "Longhorn", the stack is protected against execution by default. It must be said, however, that upon the first attempt at executing machine code in the stack or somewhere near it the system displays a dialog box allowing you to disable the protection or kill the program that violates the limitation.

To achieve the formulated goal, it is necessary to do the following:

- Insert the machine code of the custom function above the stack top;
- Save the current value of the `EIP` register and `FLAGS` register (the stack can be used for this purpose);
- Save all registers modified by the custom function;
- Set `EIP` to point to the start of the custom function;
- Pass the arguments (such as through the registers);
- Execute the function and return the result through `EAX;`
- Analyze the returned value;
- Restore the modified registers;
- Restore the `EIP` and `FLAGS` registers;
- Continue normal execution of the program

This sounds intimidating, however, the task is not as difficult as it might appear. First, try to execute the `XOR EAX,EAX/RET` function. Is it possible to convert it to machine code? It is possible to use HIEW or even FASM; however, it is also possible to carry out the same operation without exiting SoftIce. To achieve this goal, it is enough to move to any free memory area and issue the `a` (`assemble`) command. Before doing this, it is necessary to make sure that you are in the context of the application being debugged (its name is displayed in the bottom right corner of the screen) and not in the kernel; otherwise, the system would crash. The process of assembling a function in SoftIce is demonstrated in Listing 10.8.

Listing 10.8. Assembling a custom function in SoftIce

```
:a esp-10
0023:0012B0DC xor eax,eax
0023:0012B0DE ret
0023:0012B0DF

:d esp-10
0023:0012B0DC 33 C0 C3 00 DB 80 FB 77-88 AE F8 77 FF FF FF FF  3.....w...w....
0023:0012B0EC 31 D8 43 00 E8 59 48 00-00 00 00 C0 03 00 00 00  1.C..YH.........
```

Now the program lies in the stack. It only remains to execute it. How is it possible to do this? If you issue the `G esp-10` command (go to address `esp-10`), the processor won't easily cope with this task. To return control to the current location within the program being debugged, it is necessary to first save the content of the `EIP` register, which is not an easy task. The `E (esp-10) EIP` command won't help, because this command does not allow you to use expressions (recall that register names are expressions). So, the attempt at executing such a command will result in a syntax error. What can be done about this?

Try to use the `M` (`move`) command that copies memory blocks from address to address. You will be able to save a fragment of the original program on the stack and modify that program at your discretion. In particular, it is necessary to write `PUSH EAX/MOV EAX,ESP/SUB EAX,10h/CALL EAX`, or something of the sort. Briefly, you need the `CALL ESP-N` command. Because there is no such command in the command set of x86 processors, you will have to emulate it using math transformations with any additional register, such as `EAX`. In machine code, this will appear as follows: `50h/8Bh C4h/83h E8h 10h/FFh D0h`.

Now copy the fragment of the program being debugged to the stack: `M EIP L 10 ESP-20`. Here, `ESP-20` is the target address above the stack top, which does not overwrite the machine program under consideration. Now modify the neighborhood of the program being debugged: `ED EIP 83C48B50; ED EIP+4 D0FF10E8`. As you can see, this is the same code written in reverse order (this is because in x86 processors the least significant byte is located at lower address.

At this point, the preparations are complete. Now issue the `T` (`TRACE`) command four times before entering the custom function, then issue the `P RET` command to exit from there. That's all. Now the `EAX` register contains zero. The custom function has completed its operation and returned the required value. Isn't this interesting? You execute the machine code written from scratch directly within the debugger.

Another problem remains. How do you analyze the return value in the debugger? The direct approach, such as `IF (EAX==0) DO xxxx`, fails because SoftIce does not allow conditional commands. Thus, the `IF` keyword can be encountered only in breakpoints. But there is a workaround. Create a fictitious breakpoint that always becomes activated (Listing 10.9).

Listing 10.9. A fictitious breakpoint allows use of the IF keyword

```
BPX EIP IF (EAX==0) DO xxxx
```

Regardless of whether or not the breakpoint is activated, you need to restore the `EAX` register. After saving the registers, it is necessary to return the fragment of the original program to its initial location and delete the fictitious breakpoint (because the number of available breakpoints is limited). The `EAX` register can be restored using the `POP EAX` command that follows `CALL EAX`. To return the program to its initial location, use the following construct: `M ESP-20 L 10 EIP-9`. Where does `EIP-9` come from? Why is it not simply `EIP`? This is because the `EIP` value has changed in the course of the patch execution. The `9` stands for the size of the patch with the `POP EAX` command. Now it remains to issue the `R EIP = EIP-9` command to return `EIP` to the original location and continue program execution. If everything was done correctly and no protection mechanism used the uncommitted stack, the program being debugged won't crash.

Note

Do not forget about the `FLAGS` register, which also must be saved. The procedure of saving this register is the same as that for the `EAX` register, so it is not described here.

By the way, under Windows 9*x* there is still a certain probability of crashes because this system actively fills the stack with garbage. To prevent crashes, it is necessary to raise the `ESP` register slightly for the time it takes to execute all required operations and then return it to its initial location.

There is no need to enter machine codes manually all the time. This is a dull and tedious job. This is where macros come into play. Issue the `'MACRO MACRO_NAME = "xxxxx"` command and save the macro to the list of resident macros. To achieve this, start Symbol Loader, select the **Edit** | **SoftIce Initialization Settings**, go to the **Macro Definitions** tab, click the **Add** button, and assign the name and definition to the macro. Now the macro will be loaded automatically with SoftIce. You can create a library of custom extended conditional breakpoints supporting such functions as searching for substrings or comparison using wildcards such as `'*'` and `'?'`. This is a task that can be carried out in practice. If you make this effort, the power of SoftIce will grow considerably. In addition, this is an excellent chance to practice programming in machine codes.

There is another problem that can be solved using macros. SoftIce does not support nested breakpoints, which are impossible to do without (recall that for the analysis of the contents of the `EAX` register, a fictitious breakpoint had to be created). If you try to write something like `BPX CreateFileA DO "xxx; bpx EIP DO "XXXX"; x;"`, you won't achieve a positive result. SoftIce will be confused by the quotation marks and refuse to interpret such a construction. However, if you implement `bpx EIP DO "XXXX"` as a macro named `XYZ`, for example, then SoftIce will favorably accept the following construct: `BPX CreateFileA DO "xxx; XYZ; x;"`.

animated tracing in SoftIce

SoftIce lacks one useful feature available in other debuggers, such as OllyDbg. This is the possibility of step-by-step animated tracing with conditional breakpoints at each step. For example, it is possible to set a breakpoint to the `TEST EAX,EAX/Jx XXX` construct, making the debugger pop up when the `EAX` register contains a value of your choice. The construct carrying out this task might appear as `BPX IF (*word(EIP)==0xC085 && (*byte(EIP+2) & 70h)==70h)`. Here, `0xC085` is the opcode of the `TEST EAX,EAX` command, and `70h` stands for the mask of the `Jx` instruction. The entire breakpoint allows you to locate code like `if (func(1,2,3)!=0)...`. Such constructs are often encountered in protection mechanisms.

SoftIce does not correctly interpret such constructs. On the contrary, it requires the breakpoint address to be specified explicitly, for example, as follows: `BPX EIP...`. However, even in this case it creates a single breakpoint based on the current value of the `EIP` register (i.e., the value that the `EIP` register had at the moment of the breakpoint creation). It refuses to automatically "recompute" this value in the course of program execution. That's a pity! Many hackers abandon SoftIce and migrate to OllyDbg because they need such a possibility.

There is a solution that allows you to implement this capability in SoftIce: Macros allow nesting. Try the following approach: Write `MACRO XYZ="T; XYZ;"`, enter `XYZ`, and evaluate the result. SoftIce will begin to animate the program. It will do this slowly; however, the performance is enough to deal with packers and protectors. Because there is a way of animating a program being debugged, creation of conditional breakpoints no longer presents a serious problem. For example, consider the following useful macro: `MACRO XYZ = "BPX EIP;T;XYZ;"`. What does it do? It traces the program, marking the executed code, thanks to which it is possible to immediately discover which conditional jumps were executed and which were not (Fig. 10.3). Only one limitation remains: The number of breakpoints is limited; therefore, it is necessary to periodically remove them.

Thus, the following conclusion can be drawn: SoftIce is a powerful tool allowing you to achieve practically everything that you might need. The main thing here is being creative. Note that logging is not the only alternative profession of SoftIce. If desired, it is possible to create an excellent dumper on its base or anything else.

The main issue here is to grasp the idea. Although this material does not provide a ready-to-use solution, it draws your attention to the entire range of "hidden" SoftIce capabilities, which every hacker can use at his or her discretion.

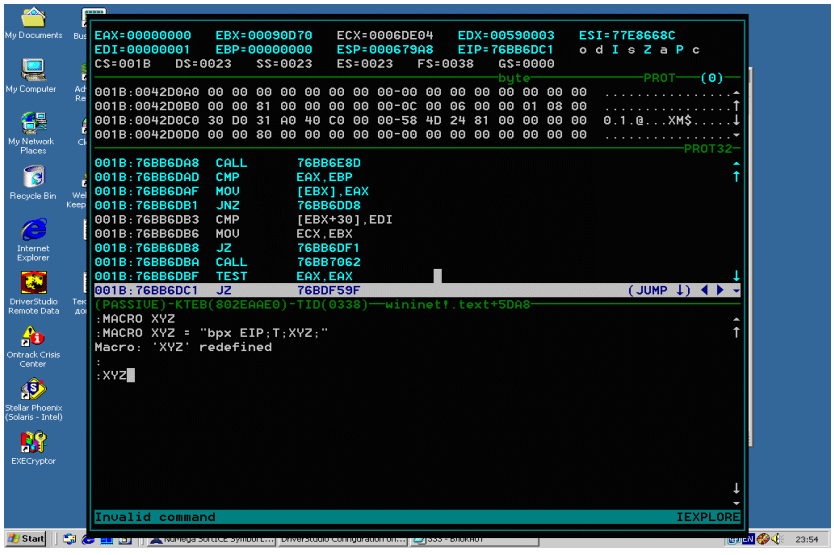


Fig. 10.3. The macro that marks executed commands

hackish tricks with arbitrary breakpoints

Breakpoints are the most powerful weapons against protection mechanisms. In the hands of a clever hacker, they can hit the target from any distance. There are three main modes of their use—break at reading, writing, or executing the memory contents located at the specified address. What can be done if the address is unknown but you know its contents? Assume that you want to set a breakpoint at the supplied password or on `JMP EAX` command used by some unpackers for passing the control to the original entry point. Or assume that you want to set a breakpoint on a sequence of machine commands (`CALL XXX/TEST EAX,EAX/JX`) corresponding to the following construct of a high-level programming language: `if (xxx(,,,) == 0)...`. The `CALL $+5 POP REG` command is typical for protection mechanisms that play tricks with self-modifying code or copy their bodies to the stack. The `PUSHFD` instruction is typically present in self-tracing programs and antidebugging protection mechanisms.

There other predictable sequences of machine commands, which typically are located in the neighborhood of the protection mechanism or even at the heart of the protection. The situation would be easy if you could set a breakpoint to a specific machine command located at a random address. Alas, the debugger needs that address and is no less interested in obtaining it than is the hacker.

Without hardware support on the part of the processor, it is impossible to solve this problem in its most general form. However, a true hacker is not going to surrender without a struggle. There are several techniques that can provide satisfactory results. This section concentrates on discussion of these techniques.

Note

A large fragment of the first edition of this book, "Identifying Key Structures of High-Level Languages," cannot be included in the second edition because of its limited volume. You will find this material on the CD supplied with this book.

secrets of step-by-step tracing

OllyDbg supports conditional breakpoints that can be set at commands (SoftIce lacks this functionality). How does it achieve this goal? It traces the program and checks one or more conditions specified by the hacker at each step. This powerful mechanism, known as the *condition to pause run trace*, allows the hacker to set a breakpoint to any combination of commands, registers, and data. Unfortunately, this mechanism is not free from limitations and side effects. In the tracing mode, program execution slows tremendously. In addition, the tracing is easy to discover, and most protection mechanisms are capable of doing this.

Among antidebugging techniques, the most popular is reading the `FLAGS` register using the `PUSHFD/POP REG` commands followed by checking the trace flag (`TF`). The Microsoft Visual Studio Debugger discloses its presence immediately, but defeating OllyDbg is not that simple. If it detects the `PUSHFD` instruction in the course of program tracing, it resets the trace flag immediately after the execution of that instruction, thus concealing its presence. However, this trick has a negative effect on self-tracing programs. There are other antidebugging techniques that OllyDbg cannot overcome. Therefore, the hacker has to neutralize antidebugging code manually. To achieve this goal, it is necessary to know how to set breakpoints at predefined combinations of machine commands. Antidebugging techniques are few, and they are well known both to hackers and to developers of protection mechanisms. So, hackers will overcome any protection mechanism, however sophisticated it might be, if protection developers do not introduce any revolutionary innovations.

setting a breakpoint at a single command

Consider a simple demonstration example. Start a simple tf.exe demo application and try to set a breakpoint to the `PUSHFD` command. Load the program into the debugger, select the **Set condition** command from the **Debug** menu, and press <CTRL>+<T>.

The **Condition to pause run trace** dialog box (Fig. 10.4) will appear, allowing you to stop tracing when the `EIP` register falls within the specified range (the **EIP is in range** option) or goes beyond the specified range (the **EIP is outside the range** option). Also supported is stopping the tracing if a certain condition is satisfied (the **Condition is TRUE** option) or if one of the predefined commands is executed (the **Command is one of** option). The latter option is the one that you need.

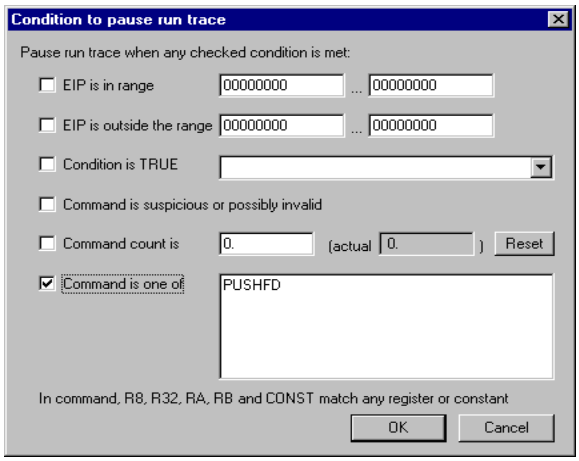


Fig. 10.4. The **Condition to pause run trace** dialog box of OllyDbg lets you set breakpoints to arbitrary machine commands

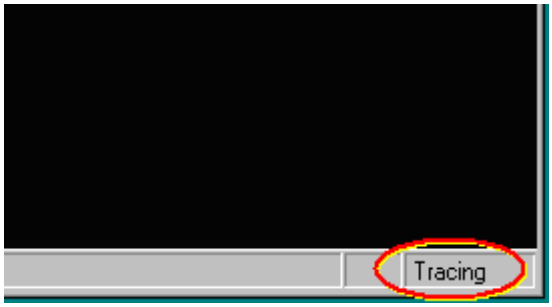


Fig. 10.5. The status bar displays the string informing you that the debugger is running in the tracing mode

Set the **Command is one of** option, and enter the `PUSHFD` command into the edit field to the right from the checkbox, then click **OK**. The dialog box will close, and you will return to OllyDbg. But if you press <F9> (run), no result will be produced because conditional breakpoints work only in the tracing mode. So, press <CTRL> then <F11> (**Debug** | **Trace into**). The debugger screen will remain unchanged so that at first glance it might seem that nothing has happened. However, in the right corner in the status bar at the bottom of the screen, the `Tracing` string will appear, confirming that tracing is in progress (Fig. 10.5).

By the way, tracing considerably overloads the operating system kernel. If you press <CTRL>+<SHIFT>+<ESC> in the course of tracing (this keyboard shortcut opens the **Windows Task Manager** dialog box), you will see that the **Kernel Times** line signals an unusually high level of the debugger activity (Fig. 10.6). If the **Kernel Times** line is not displayed, select the **Show Kernel Times** option from the **View** menu.

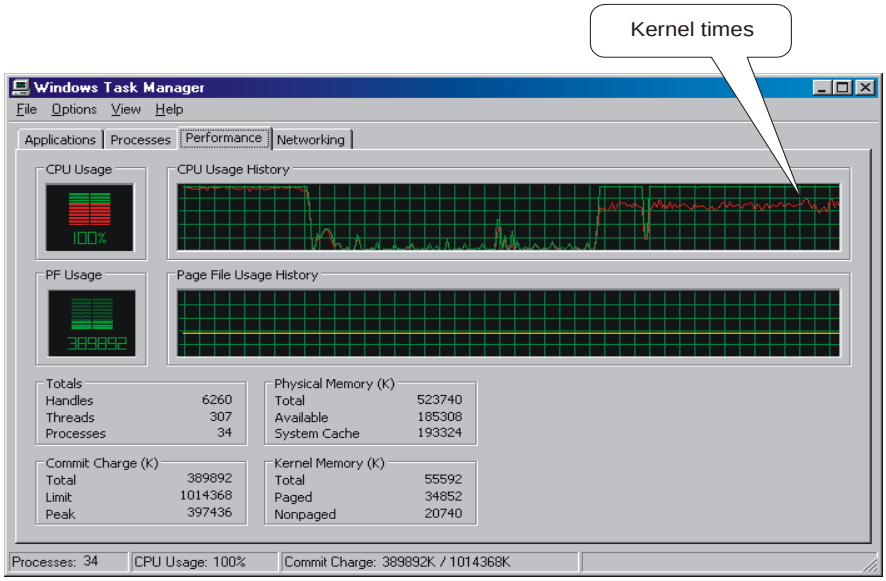


Fig. 10.6. When in the tracing mode, OllyDbg places a considerable workload on the operating system kernel

If instead of <CTRL>+<F11> you press <CTRL>+<F7> (**Animate into**), the tracing speed will be reduced tenfold. OllyDbg will refresh the CPU window at any step and highlight the current command (Fig. 10.7). This looks fine (viewing loops is especially interesting); however, this mode is not suitable for carrying out practical tasks. To stop tracing, both normal and animated, press <ESC>, and the debugger will stop at the last executed instruction.

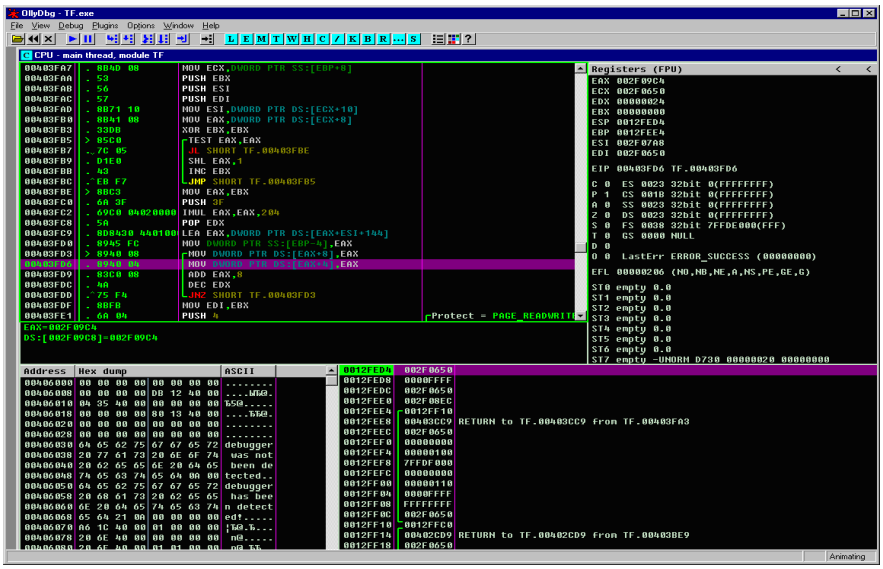


Fig. 10.7. Animated tracing in OllyDbg (the Animating string is in the right corner in the status bar)

If you continue tracing, after some time (mainly depending on the power of your CPU) the debugger will reach the PUSHFD instruction (Fig. 10.8). This is the heart of the protection mechanism that you need to analyze and neutralize. Antidebugging code is extremely simple. It takes only four lines of code (Listing 10.10)

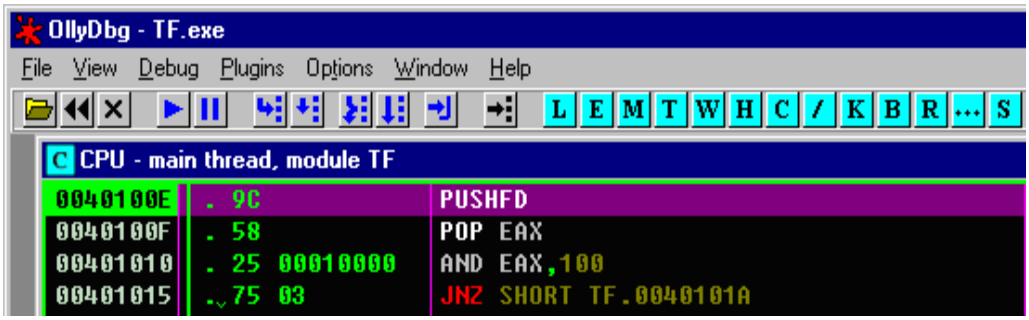


Fig. 10.8. The debugger stops when it reaches the PUSHFD command

Listing 10.10. Antidebugging code based on reading the FLAGS register

```
0040100E |. 9C              PUSHFD
0040100F |. 58              POP EAX
00401010 |. 25 00010000     AND EAX,100
00401015 |. 75 03           JNZ SHORT TF.0040101A
```

The protection pushes the FLAGS register into the stack using the PUSHFD command and immediately pops it into the EAX register, checking the trace flag (TF) with the AND EAX 100h logical operation. However, there will be no TF in the stack, because OllyDbg will automatically reset it. If you want the program to operate under other debuggers, it is necessary to replace JNZ with NOP/NOP or AND EAX, 100h with AND EAX, 0h.

If you trace the program for a while, you will exit the antidebugging procedure and find yourself within machine code typical for most high-level programs. This machine code tests the value returned by the function using the TEST EAX,EAX/Jx instruction pair (Listing 10.11).

Listing 10.11. Processing the result of the function operation

```
0040102A |. E8 D1FFFFFF    CALL TF.00401000
0040102F |. 85C0            TEST EAX,EAX
00401031 |. 74 0F           JE SHORT TF.00401042
00401033 |. 68 30604000     PUSH TF.00406030; ASCII "debugger was not been detected"
00401038 |. E8 6C000000     CALL TF.004010A9
0040103D |. 83C4 04         ADD ESP,4
00401040 |. EB 0D           JMP SHORT TF.0040104F
00401042 |> 68 50604000     PUSH TF.00406050 ;  ASCII "debugger has been detected!"
00401047 |. E8 5D000000     CALL TF.004010A9
0040104C |. 83C4 04         ADD ESP,4
```

Now try to set the breakpoint on CALL XXX\TEST EAX,EAX\JX command combination. You will fail! The **Condition to pause run trace** window clearly states that the command must be "one of" the listed commands. This means that if you enter call const; test eax,eax; jCC const, the debugger will stop on each of the listed commands, which does not correspond to your intentions. Nevertheless, this corresponds to the OllyDbg syntax rules.

It is necessary to mention other issues related to the syntax. The debugger supports templates allowing you to combine simple regular expressions. For example, R32 designates any 32-bit general-purpose register, and TEST R32, R32 stops tracing when commands like TEST EAX, EAX and TEST ESI, EDX are encountered. RA stands for any general-purpose register other than RB; therefore, the template TEST RA, RA will cause the debugger to stop when encountering instructions such as TEST EAX, EAX but not when encountering instructions like TEST EAX, EBX. Accordingly, under TEST RA, RB won't cause the debugger to stop when the TEST EAX, EAX instruction is encountered. The CONST keyword designates any direct operand; for example, the MOV RA, CONST template will cause the debugger to stop when encountering instructions like MOV AL, 69h and MOV ESI, 666h, and CALL CONST will stop the debugger on any direct procedure call. The JCC CONST expression corresponds to any conditional jump and is neutral toward conditional jumps to direct addresses. The keywords supported by OllyDbg are briefly outlined in Table 10.1.

Table 10.1. *Keywords supported by OllyDbg in regular expressions for specifying breakpoints*

keyword	description
R8	Any 8-bit register (AL, BL, CL, DL, AH, BH, CH, DH)
R16	Any 16-bit register (AX, BX, CX, DX, SP, BP, SI, DI)
R32	Any 32-bit register (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI)
FPU	Any register of the math coprocessor (ST0:ST7)
MMX	Any MMX register (MM0:MM7)
CRX	Any control register (CR0:CR7)
DRX	Any debug register (DR0:DR7)
CONST	Any constant
OFFSET	Any offset (similar to constant)
JCC	Any conditional jump (JE, JC, JNGE...)
SETCC	Any conditional instructions for setting bytes (SETE, SETC, SETNGE...)
CMOVCC	Any conditional move (CMOVE, CMOVC, CMOVNGE...)

This is great! After a little practice, you will learn how to set a breakpoint to any machine command (except for addressing commands like mem, which are not supported by OllyDbg). What steps can be taken if you need combinations of several commands, not a single command?

wildcard searching in a disassembled code

Press <CTRL>+<S> or select the **Search for** | **Sequence of commands** options from the context menu of the CPU window and enter the following template: `CALL CONST\TEST EAX,EAX\JCC CONST`. Place each command on a new line. Then click the **Find** button to search for the specified sequence of machine commands in the disassembled text. OllyDbg will quickly find the specified sequence; the search will be much faster than in the course of tracing (Fig. 10.9).

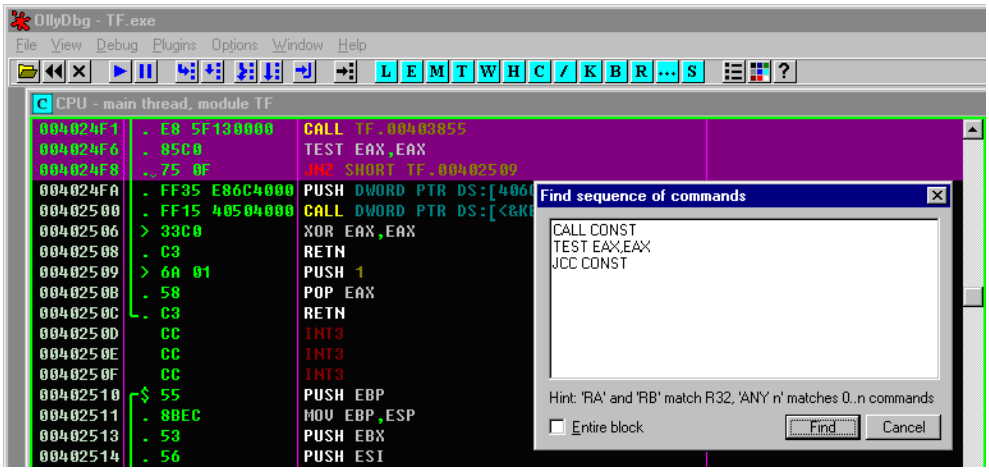


Fig. 10.9. The sequence of commands found by OllyDbg in the disassembled text

Now it is possible to press <F2> to set a conditional breakpoint to the CALL command and continue the search, marking all detected combinations. When running a program under the debugger using <F9> (note that in this case there is no need to enter the tracing mode), OllyDbg will pop up at any located CALL. Now it only remains to wait for the registration dialog box to appear (the dialog box might inform you about incorrect registration, trial period expiration, missing key file, etc.). The last CALL, which was activated before the protection started to complain, will probably be the protection mechanism spoiling the hacker's existence. This protection can be cracked by trivial inversion of the conditional jump (e.g., by replacing JNZ with JZ, or vice versa). This technique won't produce the expected result with sophisticated protection mechanisms. But it works often enough that it is expedient to add it to your armory.

Optimizing compilers can "split" the standard pattern by placing another machine command between TEST EAX, EAX instructions and conditional jumps (Listing 10.12). This increases the program performance by eliminating the pipeline idle time. However, this complicates the wildcard searching procedure.

Listing 10.12. Optimizing compiler has "split" the standard pattern

```
004013E4   |. E8 9F180000       CALL TF.00402C88
004013E9   |. 85C0              TEST EAX,EAX
004013EB   |. 59                POP ECX
004013EC   |. 8907              MOV DWORD PTR DS:[EDI],EAX
004013EE   |. 75 13             JNZ SHORT TF.00401403
```

Fortunately, OllyDbg supports an excellent search pattern, `ANY n`, which allows you to skip any number of machine commands ranging from `0` to `n`. In particular, `ANY 2` is equivalent to any two machine commands, any machine command, or no machine commands in a specific location. Thus, an improved version of the template accounting for specific features of optimizing compilers appears as follows: `CALL CONST\TEST EAX,EAX\ANY 3\JCC CONST`. The correct choice of the `n` value is extremely important. If this value is too high, there will be lots of false actuations, and a value that is too low would result in false negatives (in which case, the patterns that deserve your attention will be skipped). Practical experience has shown that the optimal value is `3`.

programming templates in machine codes

The main drawback of searching using <CTRL>+<S> is that OllyDbg searches for the combination of commands in the source disassembled text. In protected programs, the disassembled code is practically always packed or encrypted, so nothing useful can be found using static searching. It is necessary to return to step-by-step tracing and consider other conditional breakpoint mechanisms provided by OllyDbg. The **Condition is TRUE** option allows you to specify any conditions that stop tracing if the specified condition is `TRUE`. Neither templates nor regular expressions are supported here; therefore, you will have to program them manually at the level of "naked" machine code. This causes you to feel as if you have traveled back in time 15 years, when there were practically no ready-to-use cracking tools. But there is no time for nostalgic reminiscence. I strongly recommend that you open Tech Help! (**http://webpages.charter.net/danrollins/techhelp/INDEX.HTM**; the techhelp.zip file can be downloaded from many hacker sites) or an official manual on the x86 command set from Intel or AMD. The matrix of machine instructions opcodes from the Tech Help! electronic reference is shown in Fig. 10.10.

> ***Note***
> On the CD supplied with this book, you will find the "Mental Debugging" chapter from the book *Hacker Debugging Uncovered* by Kris Kaspersky, which also will be useful for understanding the principles of programming in machine codes.

Specifically, attention must be drawn to mnemonics of instructions such as `CALL CONST`, `TEST EAX, EAX`, and `JNZ/JZ`. In particular, the `CALL` command has the `E8h` opcode, which must be followed by 4 bytes of the relative address. The `TEST EAX, EAX` command in machine code appears as `85h C0h`, and `JZ/JNZ` commands are represented by the well-known 2-byte instructions `74h XXh/75h XXh`, where `XXh` is a relative offset of the jump address counted from the command start.

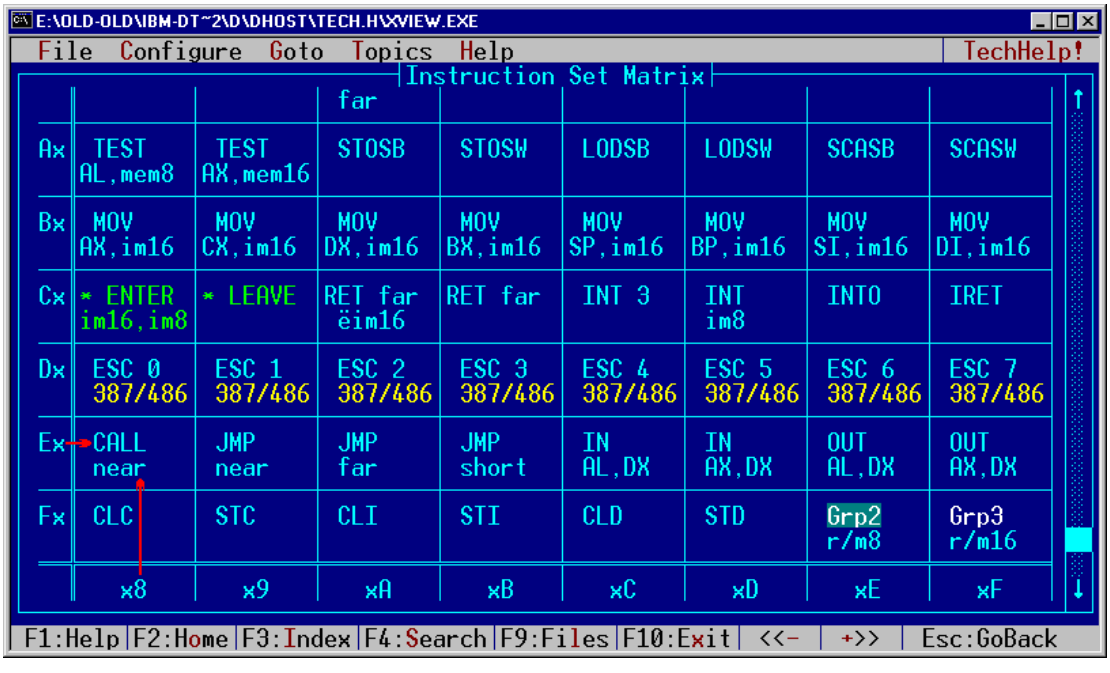


Fig. 10.10. The matrix of opcodes of machine instructions in the Tech Help! electronic reference

Having summarized all previously mentioned issues, it is possible to compose an expression activated at a specific predefined sequence of commands. Consider the contents of the `EIP` register. If it contains the value `E8h`, then this is the `CALL const` command. If you increase `EIP` by 5 bytes (which corresponds to the length of the `CALL const` command) you will get the pointer to the next command. When you compare it to the opcode of the `TEST EAX, EAX` instruction, it will be equal to `85h C0h` (`C085h`), taking into account the reverse byte order in x86. Now it only remains to check whether the third command is a conditional jump. Increase `EIP` by 2 bytes (the length of the `TEST EAX, EAX` command) and see whether this value is equal to `74h` or `75h`. If it is, then the desired sequence of commands has been found.

To "feed" this construct to the debugger, it is necessary to study the syntax of the **Condition is TRUE** option string described in the "Evaluation of Expressions" section in the OllyDbg.hlp file. This syntax considerably differs from that of SoftIce or C programming language. The OllyDbg syntax is a kind of combination of assembly, Pascal, and C languages. Equality and inequality are designated the same way as in C: `==` and `!=`, respectively. Logical operations `AND` and `OR` are designated by operators `&` and `|` (unlike in C). The expression enclosed in square brackets returns the contents of the memory cell located at the specified address, for example: `[EAX]`. By default, the cell size is 4 bytes. For type casting, the keywords `BYTE`, `WORD`, and `DWORD` are used, for example: `[BYTE ESI] == 41`. By default, all numbers are interpreted as hex numbers and must start from a digit. This means that `0FA` is a valid number and `FA` is a number written with a syntax error. Unfortunately, OllyDbg does not output error messages, thus complicating the process of debugging expressions. If a number is followed by a dot, this number is interpreted as a decimal number; therefore, the following expression is `TRUE`: (`10==16.`). Literals are enclosed into single quotation marks (`'A'==41`), and strings must be enclosed in double quotation marks. Expressions such as `[ESI]=="password"` becomes true when the `ESI` register points to the "password" ASCII string (note that square brackets are optional). If the string is specified in Unicode, use the `UNICODE` keyword, for example: `UNICODE [ESI]=="password"`. The syntax also allows arithmetic expressions like `*`, `+`, `-`, `>`, and `<`.

These rules allow you to write any template, however sophisticated it might be. In particular, searching for the `CALL const/TEST EAX, EAX/Jx const` sequence of commands can be carried out as shown in Listing 10.13.

Listing 10.13. Setting a breakpoint to the CALL const/TEST EAX, EAX/Jx const sequence

```
; ← CALL const →   ← TEST EAX, EAX →   ←   JZ const OR JNZ const   →
([BYTE EIP]==0E8) & ([WORD EIP+5]==0C085) & (([BYTE EIP+7]==74)|([BYTE EIP+7]==75))
```

Press <CTRL>+<T>, then enter this expression into the edit field next to the **Condition is TRUE** checkbox. To start tracing, press <CTRL>+<F11> or <CTRL>+<F7>. If the supplied expression is correct (note that this is rare on the first attempt), OllyDbg will stop before the entry point of the `CALL const` function (Fig. 10.11).

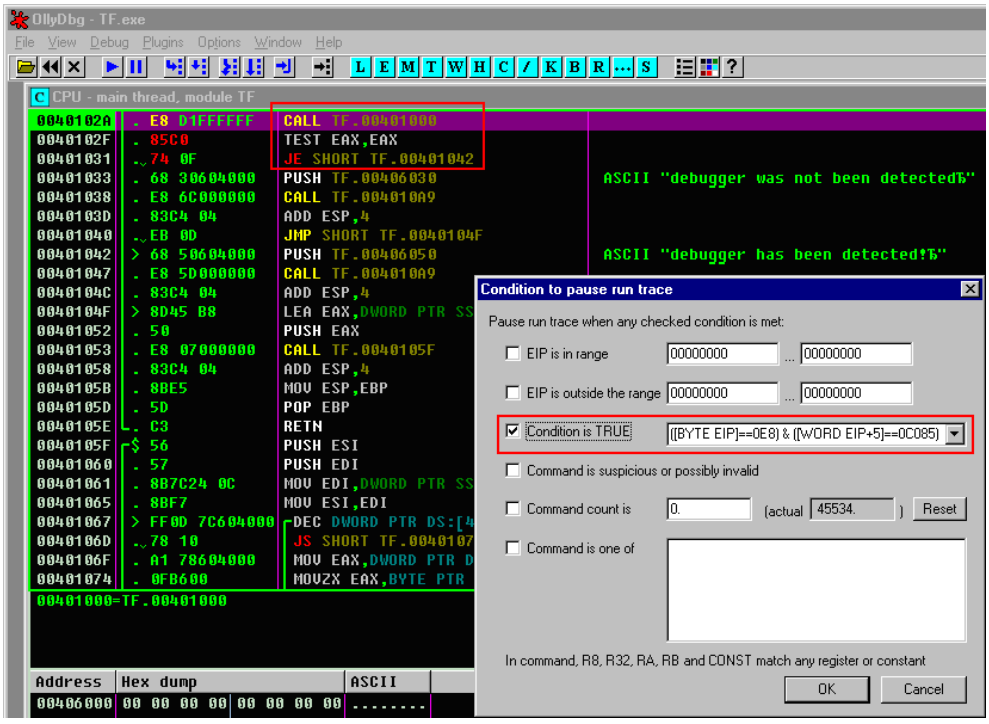


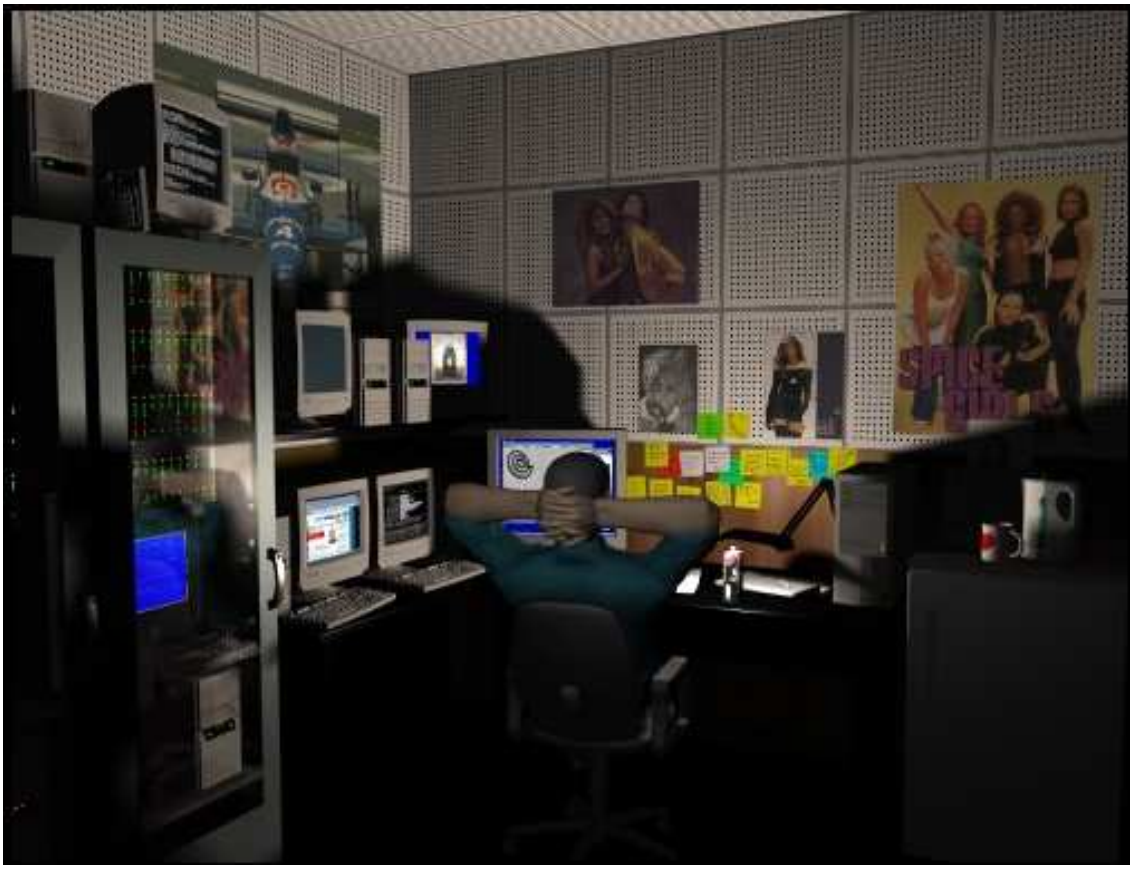
Fig. 10.11. Setting breakpoints to the sequence of machine commands

Note that it is difficult to compose complex expressions directly in the edit field. Beginners often are confused when counting braces and constantly forget what has been already written and what hasn't. Finally, the composed and debugged expressions must be saved and supplied with comments explaining what a specific expression does. Unfortunately, OllyDbg does not provide these capabilities; therefore, I recommend that you compose expressions in your favorite editor (e.g., the FAR Manager built-in editor called by <F4> with the colorer plug-in[1] allowing you to navigate among braces and control their nesting). The comments can be stored in the same way. Nevertheless, OllyDbg interprets complicated expressions too slowly, so the tracing becomes inefficient. Recall that to achieve success it is necessary to implement an analogue of the `ANY` keyword, because, as was already mentioned, optimizing compilers can insert other instructions between the `TEST EAX, EAX` and `Jx` commands.

Implementation of the `ANY` keyword is a difficult task that can be carried out only by writing a custom command-length disassembler. The most efficient approach to implementing such projects is to write them in C and integrate them with the debugger as plug-ins. The source code of the disassembler is supplied with OllyDbg (**http://www.ollydbg.de/srcdescr.htm**), which means that the main part of job has already been done. The remaining can be delegated to Perl. Why not? It is hard to find a better engine for composing and analyzing regular expressions. Thus, by combining Perl with the disassembler, you will obtain a powerful instrument. The flow of commands obtained by the OllyDbg tracer is supplied to the input of the plug-in, the disassembler converts it into text, and Perl searches the chains of the required commands in this text.

Here an important issue should be mentioned. To successfully deal with self-modifying code, it is necessary to abandon prefetching. Thus, the plug-in must analyze only instructions that have already been executed. OllyDbg places such instructions in the tracing log. However, analysis of polymorphous code is an overly ambitious goal. You can enjoy the victory even if you combine Perl with the disassembler and obtain the possibility of setting breakpoints at any combination of commands.

This will allow you to bypass all existing antidebugging techniques. Any antidebugging trap is specified by some combination of machine commands, and the number of possible variations is limited. Thus, when you encounter protection that crashes the debugger, it will be enough to find the location within the protection mechanism where the antidebugging technique is implemented and add this pattern to your plug-in. Finally, you will obtain an analogue of IceExt for OllyDbg, even more powerful than the actual IceExt (and slower because of the use of Perl).



[1] Colorer is a syntax highlighting and text-parsing library that provides services to parse text in an editor in real time and transform results into colored text. You can download it from **http://plugring.farmanager.com/cgi-bin/downld.cgi?Lang=Eng**.

cracking through coverage

The most difficult part of the cracking job is "locating" the protection code, which often is nothing but a trivial sequence of commands like `CALL CheckReg/TEST EAX,EAX/Jx unregistered`. After you locate this code, completing the cracking is an easy matter. A hacker's armory provides breakpoints, cross-references, and many other techniques. However, they are not always efficient. If these techniques have failed, the hacker must do a good deal of mental work to find new techniques of code investigation. This section discusses one of such method—cracking through coverage.

general idea

Consider a hypothetical program protected by trial period. This program works well during the predefined period, and displays the nag screen requiring the user to register the program, and then terminates operation after trial has expired. If you locate the code that displays that dialog box on the screen, it will only remain to correct the conditional jump or add several `NOP` instructions. However, you must first discover what must behacked. It is possible to set a breakpoint to an API function or follow cross-references to error message strings. However, these approaches are not efficient enough. There are lots of API functions responsible for reading the current date or for creating dialog boxes. And text strings are often encrypted or stored in resources.

What about comparing the program tracing logs before and after the expiration of the trial period? Before expiration, the code that displays a nag-screen is not executed; after expiration, this code is executed. Thus, the cracking is reduced to analysis of the code coverage. The coverage refers to code that gained the control at least once. The task of measuring the code coverage is delegated to profilers and their accompanying utilities. Analysis of the code coverage allows you to crack other types of protection mechanisms, such as nag screens displayed arbitrarily or periodically. To implement this approach, start the program being cracked and exit it immediately, without waiting for the nag screen to appear. During the next run, patiently wait for the nag screen to appear, and then compare the results of the code coverage. Note that this method is useless if the nag screen appears before the program starts up.

Protection mechanisms based on the key file or serial number also can be easily cracked through analysis of the code coverage if at least one valid key is available. Some users might ask: Why do you need to crack the protection if there is a valid key already? The answer is straightforward. Many programs try to check the validity of the key by using the Internet. If requests for confirmation are sent from different IP addresses, the key is placed on the black list and declared "pirate." If the registration server receives such a key again, it sends the deactivation signal to the program. The coverage allows you to quickly determine where the check takes place and to block it. In many cases, the problem can be solved by a firewall. However, some program can detect network presence earlier, for example, by of the `InternetGetConnectedState` API call. If the Internet connection is detected, the protection insolently requests the user to disable the firewall for key activation. With electronic keys, the situation is similar. By comparing the trace log with the key and without it, you will be able to see all checks that were carried out. Having located these checks, you can either kill them all or write a custom emulator and debug it in the same way—by comparing the tracing results.

Analysis of the code coverage is a reliable cracking technique that allows you to discover all checks, even if they are carried out from different locations and with different probability. After this analysis, you will obtain a program "snapshot" that allows you to easily discover everything that otherwise can be located only by long tracing or tedious disassembling.

choosing tools

There are lots of utilities intended for analysis of code coverage, both commercial and freeware. Among the undisputable leaders are Intel Coverage Tool, NuMega TrueCoverage, and Microsoft's profiler supplied as part of the Microsoft Visual Studio package. However, all these tools are oriented toward working with programs for which source code is available. This means that they are unable to process a "pure" binary file without the source code. Fortunately for hackers, it is easy to deceive such a program by generating all required information on the basis of the disassembled listing created by IDA Pro.

The profiler needs only information about line numbers and function manes. It does not work with the source code. To solve the problem, it is enough to add the debug information to the file being cracked. However, first it is necessary to decide which format it must have. Some profilers can work with a trivial map (which can be automatically generated using IDA Pro); however, most profilers work with debug information in custom formats, which usually are undocumented. As a rule, these formats of debug information rely on specific proprietary compilers (e.g., in case of Intel Coverage Tool, this is either Intel Fortran Compiler or Intel C++ Compiler). In addition, Intel Coverage Tool requires the presence of the Intel VTune Performance Analyzer (Fig. 10.12), which takes more than 200 MB and poorly operates under VMware.

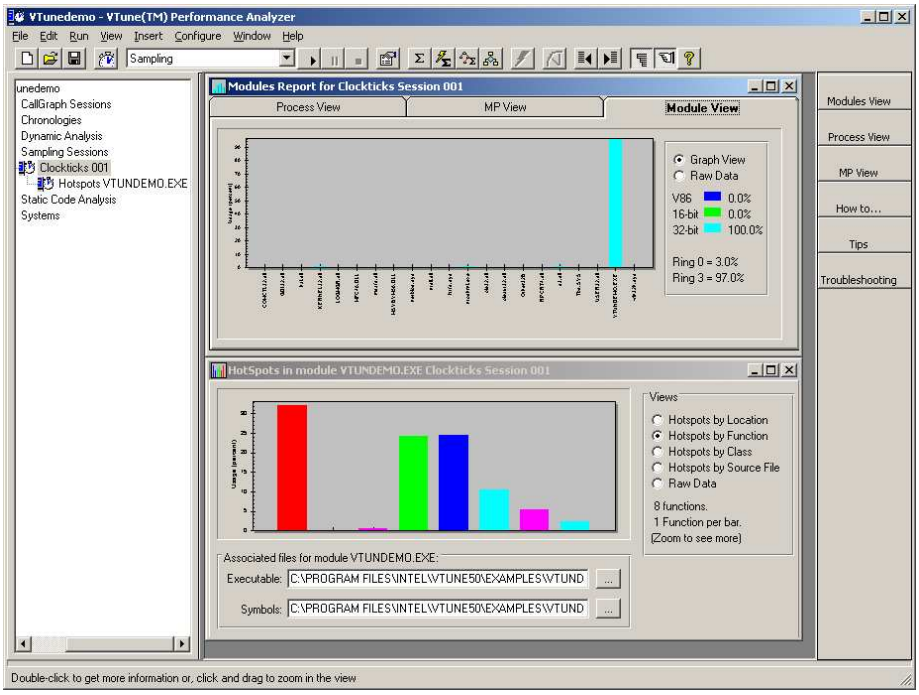


Fig. 10.12. Intel's VTune Performance Analyzer is one of the most powerful commercial profilers; it works with Intel Coverage Tool to determine the code coverage

NuMega, which has no proprietary compilers, appears more attractive in this respect. Unfortunately, the version of NuMega TrueCoverage supplied as part of Driver Studio supports only drivers (Fig. 10.13). At the moment, there is no TrueCoverage version intended for working with applications. (Before NuMega was purchased by Compuware, such a version was available.)

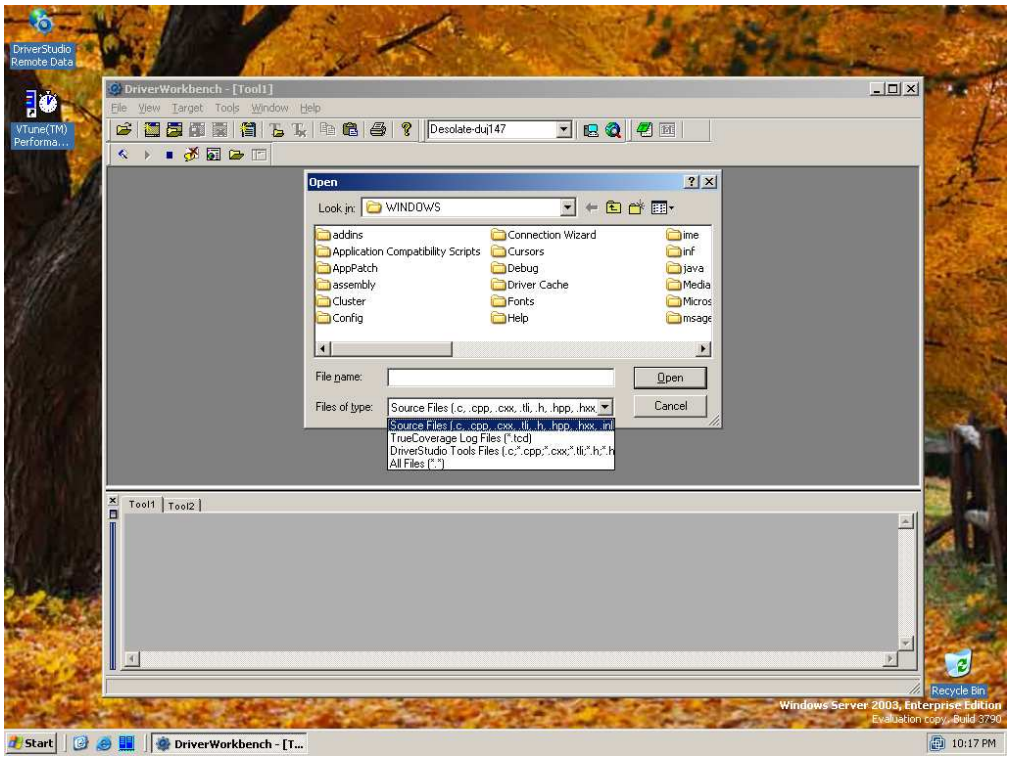


Fig. 10.13. NuMega TrueCoverage is an excellent tool for determining the code coverage; however, it is oriented only toward drivers

Microsoft profiler.exe can profile only relocatable programs (i.e., programs that do not have an empty `FIXUP TABLE`). Most binary files are not relocatable. And, although there are lots of heuristic algorithms allowing for restoring relocatable elements (in particular, they can be encountered in dumpers), the functional capabilities of Microsoft profiler.exe are not worth spending your time on this profiler (note: more details on relocatable programs will be provided in Chapter 11)

algorithms for determining coverage

There are at least two algorithms for determining the coverage. These are tracing and breakpoints. Program execution in step-by-step tracing mode produces the complete track of the program run, disclosing addresses of all executable instructions. Unfortunately, protected programs usually do not allow tracing. In addition, this approach is slow and inefficient.

Another algorithm is insertion of software breakpoints into the beginning of all machine instructions. Each breakpoint that has been activated once is removed, and the code that corresponds to it is declared covered. Most profilers operate in this way. The only difference is that they insert breakpoints into the beginning of the group of instructions representing specific lines of the source code (which is why they need information about lines). The operating rate is better than with the previous approach because of the removal of breakpoints. This is especially true for loops. The drawback of this approach is that software breakpoints can be easily discovered; such a breakpoint is the `CCh` byte written over the instruction and detectable by recomputing the checksum. This makes it unsuitable for analysis of protection mechanisms.

Unfortunately, there are only four hardware breakpoints; therefore, the only reliable method of determining the coverage is full processor emulation. Note that there is no need to develop an emulator from scratch. It is possible to use a freeware emulator distributed with the source code (e.g., Bochs). As a variant, it is possible to use a rough method of determining the coverage by pages. When using this method, all pages of the process are marked unavailable and then exceptions are caught by which it is possible to determine the coverage. If the protection code is located in a separate procedure (which usually is the case), it can be detected. However, there also is the probability of failure to discover the protection procedure. Everything depends on the space occupied by the protection code. This method does not detect standalone conditional jumps. However, if desired, this method could be improved.

As mentioned earlier, the improved algorithm marks all pages unavailable. However, in case of exception, the `PAGE_NOACCESS` attribute is not removed. Instead, the `EIP` register is read, after which it is necessary to decode the machine instruction and set a hardware breakpoint after its end. If the instruction in question is a conditional jump, you must either analyze flags that determine the conditions of its actuation or set two hardware breakpoints: one after the end of the jump and one at the address pointed at by the jump. Then, temporarily remove the `PAGE_NOACCESS` attribute, execute the instruction on a "live" processor, mark it as covered, and restore the `PAGE_NOACCESS` attribute. This attribute must be permanently removed only when all instructions belonging to the current page are covered. This approach ensures a satisfactory operating rate. In addition, it is easy to implement because it requires you to implement only the instruction length disassembler. Note that although the program being cracked can easily detect both hardware breakpoints and `PAGE_NOACCESS` attributes, most protection mechanisms do not indulge in this.

choosing an approach

Although development of a custom profiler requires considerable time and effort, at first it is possible to use the debugger supporting tracing into the log and bypassing most typical protection mechanisms. The famous OllyDbg is suitable for this role. There are lots of plug-ins for this debugger that conceal its presence from most protection mechanisms.

By starting the program before and after protection activation, you will produce two tracing logs. Read addresses of machine commands, load it into the array of the custom log, sort it, remove repetitions, and then compare both arrays. The comparison is reduced to searching, whose fastest implementation in a sorted array is achieved by using the fork method. With all this being so, it is necessary to compare not only the first array to the second one but also the second array to the first one. For convenience, it is useful to output mnemonics of machine commands near the addresses. These mnemonics can be easily retrieved from the log.

Listing 10.14 provides a simplified version of the log analyzer, which does not use sorting and carries out the comparison by sequential searching. Therefore, the processing of large logs is lengthy (although, taking into account the power of a contemporary processor, the time required for completion of this process won't be too long).

Listing 10.14. The log-coverage-diff.c program comparing the OllyDbg tracing protocols

```
#define XL      1024              // Maximum length of one line of the log.
#define NX      (1024*1024)       // Maximum number of lines in the log.

// Add new address to the array,
// provided that it was not encountered earlier.
addnew(unsigned int *p, unsigned int x)
{
    int a; for (a = 1; a<*p; a++) if (p[a] == x) return 0; if (a == NX) return -1;
        p[0]++; p[a] = x; return 1;
}

// Output the results to the screen.
PRINT(unsigned int x, FILE *f)
{
    char *z; char buf[XL];
    while(fgets(buf,XL-1,f)) if((strtol(buf,&z,16)==x)&&(printf("%s",buf)|1))break;
}

// Compare two arrays of addresses.
diff(unsigned int *p1, unsigned int *p2, FILE *f)
{
    int a, b, flag;
    for (a = 1; a<*p1; a++)
    {
        for(b = 1, flag = 0;b<*p2; b++) if ((p1[a]==p2[b]) && ++flag) break;
        if (!flag) PRINT(p1[a],f);
    }
}

main(int c, char **v)
{
    int f=0; char buf[XL]; FILE *f1, *f2; unsigned int *p1, *p2; char *x;
    if (c < 3) return printf("USAGE: log-coverage-diff.exe file1 file2\n");
    p1 = (unsigned int*) malloc(NX*4); p2 = (unsigned int*) malloc(NX*4);
    f1 = fopen(v[1],"rb");if (!f1) return printf("-ERR: open %s\x7\n",v[1]);
    f2 = fopen(v[2],"rb");if (!f2) return printf("-ERR: open %s\x7\n",v[2]);
    fgets(buf, 1023, f1); fgets(buf, 1023, f2);

    while(f<2 && !(f=0))
    {
        if (fgets(buf, XL-1, f1)) addnew(p1,strtol(buf,&x,16)); else f++;
        if (fgets(buf, XL-1, f2)) addnew(p2,strtol(buf,&x,16)); else f++;
    }

    if (fseek(f1, 0, SEEK_SET)) return printf("-ERR: seek %s\x7\n",v[1]);
    if (fseek(f2, 0, SEEK_SET)) return printf("-ERR: seek %s\x7\n",v[2]);

    printf("\ndiff p1 -> p2\n");diff(p1,p2,f1);
    printf("\ndiff p2 -> p1\n");diff(p2,p1,f2);

    return 0;
}
```

This program is compiled as usual—with the default command-line options. In particular, when using the Microsoft Visual C++ compiler, the command line appears as follows: `cl.exe log-coverage-diff.c`. If you need to create an optimized variant, use the following command line: `cl.exe /Ox log-coverage-diff.c`. Compiling from the command line is completed normally. The source code and executable files for this example can be found at the CD supplied with this book.

Note

If you select the entire text in Listing 10.14, copy it into the integrated development environment (IDE), and then press <F7> (Build), do not be surprised that the program is not compiled and built successfully. It will not compile because by default Microsoft Visual Studio creates a C++ project and this program is written in classical C.

To illustrate the concepts explained in this section, consider an example of a simple protection by trial period and try to crack it by comparing the coverage before and after expiration of the period. The source code of the program being cracked can appear as shown in Listing 10.15. The trial period is specified by the following constants: `ye` (year), `me` (month), and `md` (day). You must reset these constants according to the current date so that the program is still usable for your experiments.

Listing 10.15. The listing of the coverage.c program protected by a trial period

```
int expired = 0;

#define md 3
#define me 8
#define ye 2007

foo()
{
        SYSTEMTIME sy; GetSystemTime(&sy);
        if ((sy.wYear&0xF)*sy.wMonth*sy.wDay > (ye&0xF)*me*md) expired=1;
}

main()
{
        printf("coverage trial\n");
        foo();
        if (expired) printf("trial has expired!\n"); else printf("trial ok\n");
}
```

Compile the program as in the previous example (see Listing 10.14). As a result of the compiling and building process, the coverage.exe file will be created. Start this file for execution to make sure that the program outputs the `trial ok` string. Then modify the system data so that the current date is beyond the trial period and start the program again. Now it will display the `trial has expired!` string.

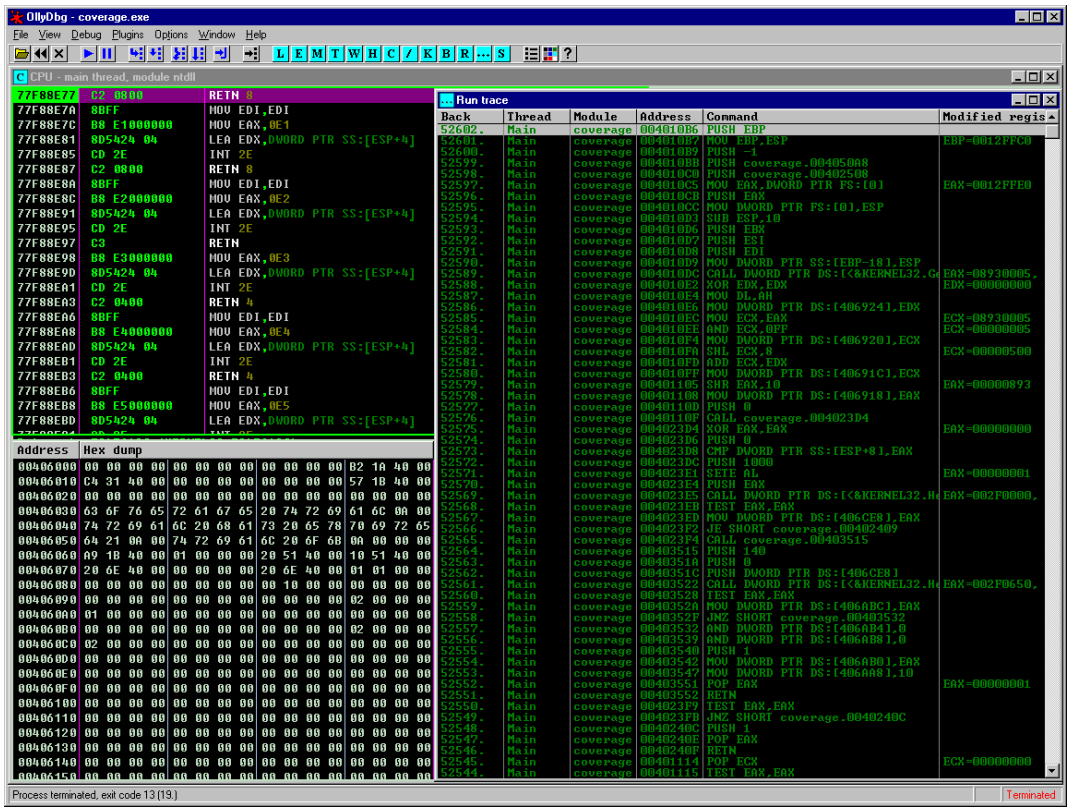


Fig. 10.14. Determining the code coverage using OllyDbg

Restore the previous data and load coverage.exe into OllyDbg (Fig. 10.14). From the **View** menu choose the **Run trace** command, then press <SHIFT>+<F10> to call the context menu, and choose the **Log to file** command from that menu. A dialog box will appear in which it is necessary to place the log file name, for example, coverage1.txt. Then press <CTRL>+<F11> (Trace Into) and wait for the program to complete its operation. A coverage1.txt file approximately 3 MB in size will be created. Then, in the **Run trace** window, press <SHIFT>+<F10> and close the log file.

Press <CTRL>+<F2> to restart the program and change the system date to a later one. Then return to the **Run Trace** window, press <SHIFT>+<F10>, and supply the file name (e.g., coverage2.txt). Start tracing by pressing <CTRL>+<F11>, wait for the program to complete, and close the log file by pressing <SHIFT>+<F10>.

Now you have two log files that can be processed by the log-coverage-diff.exe utility. The results of this operation, allowing you to view the difference between code coverage variants before and after the expiration of the trial version, are shown in Listing 10.16.

Listing 10.16. Comparing program code coverage before and after trial period expiration

```
$log-coverage-diff.exe coverage1.txt coverage2.txt
diff p1 -> p2
00401076 Main    PUSH coverage.00406054
0040107B Main    CALL coverage.00401085
00401080 Main    ADD ESP,4

diff p2 -> p1
0040103B Main    MOV DWORD PTR DS:[4068F0],1
00401067 Main    PUSH coverage.00406040
0040106C Main    CALL coverage.00401085
00401071 Main    ADD ESP,4
00401074 Main    JMP SHORT coverage.00401083
```

Having analyzed the results shown in Listing 10.16, even a beginner could guess that the double word located at address `4068F0h` is the global flag determining whether the trial version has expired. The `MOV DWORD PTR DS:[4068F0], 1` command is the one that sets this flag when the trial period expires. Thus, to neutralize this protection, it is enough to replace `MOV DWORD PTR DS:[4068F0], 1` with `MOV DWORD PTR DS:[4068F0], 0`.

Load coverage.exe into HIEW, press <ENTER> two times to switch to the disassembler mode, then press <F5> (Goto), and enter the address of the `MOV` command preceding it with the dot to inform HIEW that this is the address, not the offset from the start of the file (`.40103B`). Press <F3> to activate the editing mode, then press <ENTER> to input the assembly command (Fig. 10.15). HIEW will automatically copy the current instruction to the edit line. It only remains to replace `1` with `0` and press <F9> to save modifications. After this, it will be possible to start the file independently on the current date.

As a variant, it is possible to open the coverage.exe file in IDA Pro and view the contents of memory cells located at and near the addresses output by the log-coverage-diff.exe utility (Listing 10.17; the differences in code coverage before and after the expiration of the trial version are in bold font). This will probably help you find a faster method of cracking.

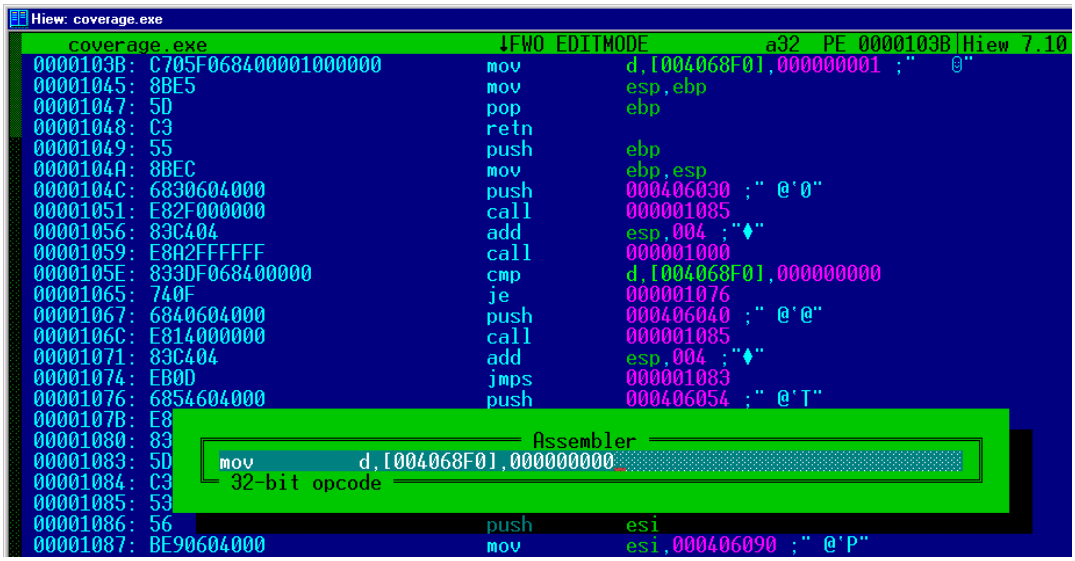


Fig. 10.15. Hacking the coverage.exe program in HIEW

Listing 10.17. A fragment of the disassembled listing of the coverage.exe program

```
0401000 sub_401000    proc near
0401000               push    ebp
0401001               mov     ebp, esp
0401003               sub     esp, 10h
0401006               lea     eax, [ebp+SystemTime]
0401009               push    eax                     ; lpSystemTime
040100A               call    ds:GetSystemTime
0401010               mov     ecx, dword ptr [ebp+SystemTime.wYear]
0401013               and     ecx, 0FFFFh
0401019               and     ecx, 0Fh
040101C               mov     edx, dword ptr [ebp+SystemTime.wMonth]
040101F               and     edx, 0FFFFh
0401025               imul    ecx, edx
0401028               mov     eax, dword ptr [ebp+SystemTime.wDay]
040102B               and     eax, 0FFFFh
0401030               imul    ecx, eax
0401033               cmp     ecx, 90h
0401039               jle     short loc_401045
040103B               mov     dword_4068F0, 1
0401045 loc_401045:
0401045               mov     esp, ebp
0401047               pop     ebp
0401048               retn
0401048 sub_401000    endp
0401048
0401049 _main    proc near
0401049               push    ebp
040104A               mov     ebp, esp
040104C               push    offset aCoverageTrial ; "coverage trial\n"
0401051               call    _printf
0401056               add     esp, 4
0401059               call    sub_401000
040105E               cmp     dword_4068F0, 0
0401065               jz      short loc_401076
0401067               push    offset aTrialHasExpire ; "trial has expired!\n"
040106C               call    _printf
0401071               add     esp, 4
0401074               jmp     short loc_401083
0401076 loc_401076:
0401076               push    offset aTrialOk ; "trial ok\n"
040107B               call    _printf
0401080               add     esp, 4
0401083
0401083 loc_401083:
0401083               pop     ebp
0401084               retn
```

As you can see, the heart of the protection mechanism is concentrated in the `sub_401000` procedure that compares the current date to a hardcoded constant, after which it executes the `jle` conditional jump to `loc_401045` (the trial period has not expired yet) or does not do anything, in which case the `mov dword_4068F0, 1` command is executed. The global flag of the trial version expiration is checked in a single location—at address `040105Eh`, after which there follows the `0401065 jz short loc_401076` conditional jump that chooses which of the two messages must be displayed on the screen. If you replace `jle short loc_401045` with `jmp short loc_401045`, the trial period expiration flag will never be set.

Note

In real protected programs, the registration flag is usually checked multiple times.

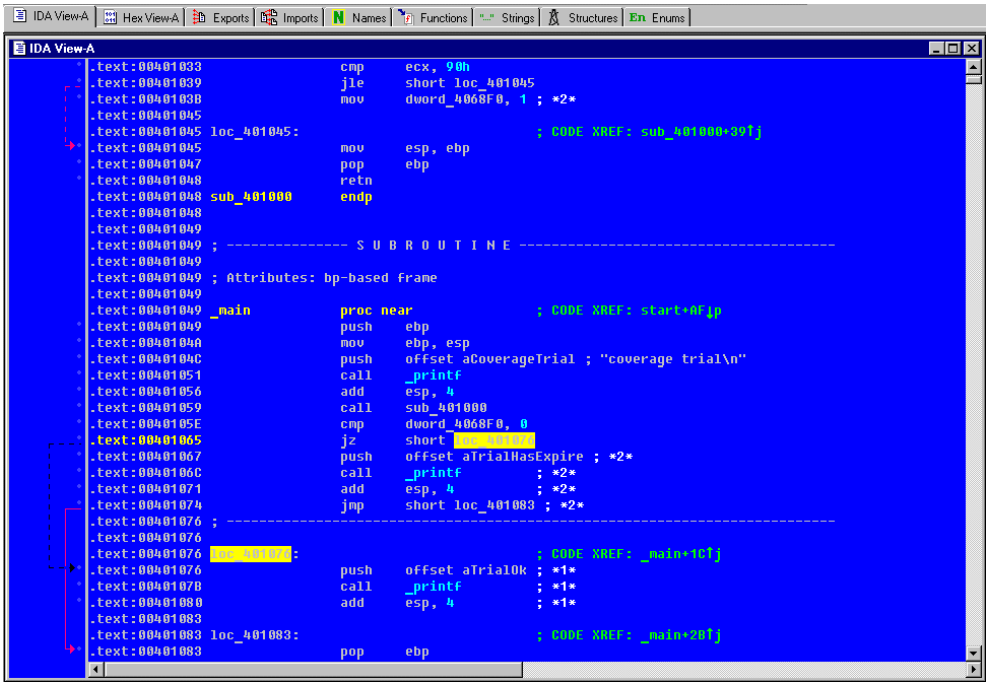


Fig. 10.16. The difference in the code coverage of the program being investigated shown in IDA Pro

To illustrate this technique, it is possible to write a simple script in the IDA C language, which reads the result of the coverage-diff.exe operation and marks the difference in coverage with some character. In the example shown in Fig. 10.16, the `*1*` mark means that the given command was executed only in the first run and `*2*` means that the command was executed only in the second run.

Note

A fragment of the first edition of this book, "IDA Comes onto the Scene," providing more detailed information on the IDA C language syntax and techniques of writing scripts, can be found on the CD supplied with this edition.

summary

In this, the concluding chapter of the second part of this book, you have become acquainted with the several interesting cracking methods and fundamental ideas that will allow you to proceed to mastering advanced techniques of code investigation.