CRACKPROOF YOUR SOFTWARE

THE BEST WAYS TO PROTECT YOUR SOFTWARE AGAINST CRACKERS

Pavol Červeň

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Crackproof Your Software—The Best Ways to Protect Your Software Against Crackers

Pavol Cerven
NO STARCH PRESS

San Francisco

1 2 3 4 5 6 7 8 9 10−05 04 03 02

Crackproof Your Software is an English version of *Cracking a jak se proti nemm brţnit*, by Pavol Cerven, the original Czech version (80−7226−382−X), copyright © 2001 by Computer Press. English translation prepared by Skrivanek Translation Services.

Publisher: William Pollock
Editorial Director: Karol Jurado
Cover and Interior Design: Octopod Studios
Composition: 1106 Design, LLC
Copyeditor: Andy Carroll
Indexer: Broccoli Information Management

Distributed to the book trade in the United States by Publishers Group West, 1700 Fourth Street, Berkeley, CA 94710; phone: 800−788−3123; fax: 510−658−1834.

Distributed to the book trade in Canada by Jacqueline Gross & Associates, Inc., One Atlantic Avenue, Suite 105, Toronto, Ontario M6K 3E7 Canada; phone: 416−531−6737; fax 416−531−4259.

For information on translations or book distributors outside the United States and Canada, please see our distributors list in the back of this book or contact No Starch Press, Inc. directly:

No Starch Press, Inc.
555 De Haro Street, Suite 250, San Francisco, CA 94107
phone: 415−863−9900; fax: 415−863−9950; info@nostarch.com; http://www.nostarch.com

Library of Congress Cataloging−in−Publication Data

Cerven, Pavol.
[Cracking a jak se proti nemm brţnit. English]
Crackproof your software/Pavol Cerven.
p. cm.

Includes index.

1−886411−79−4

1. Software protection. 2. Computer security. 3. Data protection. 4. Computer crimes. I. Title.
QA76.76.P76 C47 2002
005.8−−dc21
2002012207

ABOUT THE AUTHOR

I started programming on 8−bit computers and the only good programming language for them was assembler. My father bought a PC about four years ago, and if not for that PC, this book probably would not exist. (When I finished this book, I was 23 years old.)

I have tried several programming languages but have remained faithful to assembly because I think it is the clearest and the most beautiful programming language. What you write in assembly is exactly what you will find in the compiled version — nothing less and nothing more.

In the days of DOS I dealt with the problems closest to assembly — viruses, and even dreamt about working for an antivirus software company. When Windows 9x appeared, assembler was used less and less and there were also fewer viruses (at least fewer assembly viruses). That's when I discovered some−thing new, unexplored and often mysterious: protecting software against illegal copying. As I explored this challenge, I became so preoccupied with it that I quit the virus field (though I still enjoy the protection field and think I will stick with it for some time to come).

My page at http://www.anticracking.sk will give you a bit more information about what I do and our product, SVK − Protector: a powerful tool for securing software against unauthorized copying, use, and distribution. SVKP was designed with ease of use and high speed as a priority without sacrificing high levels of protection. It offers three different methods of securing: It uses RSA algorithm, API functions, and new anti−debug tricks.

Pavol Cerven

Introduction

This book is designed to help all programmers who have ever written their own software to better protect their software from illegal copying. It will also be useful to programmers creating freeware who wish to protect their source code.

The idea to write a book like this came to me some time ago when I realized how poorly the topic is covered and how difficult it is to acquire the information necessary to adequately protect software. When I was involved with game production in the Czech and Slovak Republics, I was astonished at how simple their protection was, and that very often they had no protection at all — yet it is so easy to protect software, at least at a basic level.

The problem lies in the lack of information and experience in this field. That's why I wrote this book, which will present many previously unaddressed topics concerning software protection.

Protection as a Deterrent

My experience tells me that there is no protection that cannot be easily removed and, as such, much of the work you will put into protecting your software is simply a deterrent, delaying the inevitable. It's only a matter of time, possibilities, and patience before a cracker cracks your software.

Of course, the better your deterrent, the more time you'll have to sell your software before you find it available (or crackable) for free, online. What creators of a program or game would want to find their product, whether shareware or commercial software, pirated on the Internet the very day of the release? That would definitely result in reduced sales.

Good software protection prevents the cracker from removing the protection correctly. With such protection, the program won't work, or won't work correctly, and more people will buy an original copy. Of course, a successful crack will appear in the course of time, but the time you buy is money earned. Really good protection will buy a considerable amount of time and will engender several versions of the crack, some of which will not work properly. In such a case, even many hardcore pirates will buy an original copy rather than try to crack one, just to avoid the hassle.

Working with Assembler

In later chapters you'll find many examples of applications protected from debugging, disassembling, or possible decompiling. The examples are all in assembler, but they are written as comprehensibly as possible and are accompanied by footnotes in a source code. Even a mediocre assembler programmer should be able to understand them. I chose not to use a higher−level language like C++ code because it wouldn't be understandable to programmers who work in Delphi, and vice versa. I chose not to use Visual Basic because most examples cannot be written in it. Assembler is the best choice because even code written in C++ will have some parts written in assembler.

Another advantage of assembler is that it can be directly inserted both into C++ and Delphi code, so assembler examples are universal for both languages. Visual Basic programmers can also insert the code into a library created in another programming language (assembler, C++, or Delphi) and then call the library from the application code. This is certainly not a perfect solution, but it is better than no protection at all.

Publishing Cracker Tricks

This book took considerable effort to write. I had to do a good bit of research, and most of what I present here comes from the web pages of crackers. There are plenty of them, and it is sad that there is almost

nothing comparable for developers.

Some people argue that information like that presented in this book should not be freely accessible to everyone. However, keeping it secret would be counterproductive. The fact is, crackers are very well informed, while developers have virtually no resources. When a cracker learns how to remove a certain kind of protection, it is only a matter of time before detailed information on how to do so is published on specialized web pages. On the other hand, developers who don't follow the field of cracking carefully will not be aware of how easily their protection can be cracked and will continue to use this protection, even though it may be removed in a matter of minutes.

It is no surprise that crackers create the best software protection, since they are often the best informed and have the most experience. This situation will hopefully change in the future, and I will be very glad if this book helps in this effort.

My thanks go to all the people without whom this book would never have been written.

First, my special thanks to my friend **Linda** and my family, who tolerated my late−night sessions and my bad mood in the mornings when I had to go to work.

Thanks to my Internet friends:

- **EliCZ** Thanks for all the help and for your faultless source code. There is hardly a better system programmer than you, really.
- **Ivan Bartek** Thanks for everything; I look forward to our future cooperation.
- **Miroslav Bambošek** You helped me a lot with your keen observations and your help with C examples. I would probably be unable to manage SafeDisc without you.
- **Ice** Thanks for everything, especially for your inspiration.
- **Stone** Your wonderful source code helped me in many cases.
- **The Owl** You are a real master of anti−debugging tricks and other hidden secrets of Windows 9x.
- **Liquid** Thanks for the anti−FrogsICE tricks.
- **Pet'o Somora** You are a wonderful mathematician and an even better friend. Thanks for your patience in explaining those problems.

Further, thanks to the following people: Hoe, Doom, HHup, Brbla, Slask, Lorian, Christopher Gabler, Nihil, Iceman, Maxx, Ender, Alfo, Sadman, Crow, Rainman, SaHo, Momo, Dugi, Ivan, Maroš, Mikie, KremeH, Neuron, Daemon, SAC, Light Druid, and Vladimir Gneushev. And to everyone whose names I have forgotten to list here but who helped with this book, thank you.

Chapter 1: Basics

Before you can protect your software well, you must first understand the methods crackers use to crack your software. *Crackers* are the people who try to remove the protection from your software so that it can be illegally distributed.

Why Crackers Crack

The first mistake developers often make is in underestimating the power and number of crackers, and that's the worst mistake any developer of protection can make. Mostly, crackers are very smart people who will work on removing software protection for days at a time, and in extreme cases even for weeks, for the challenge of it. The cracker's success almost always depends on his motivation.

It may surprise you to learn that most of the cracker's motivation is not financial. Crackers post their cracks and information for free, after all. They're not making money off your software, though the people who use their cracks are saving money. Rather than crack software for financial gain, crackers are taking part in a sort of informal competition. A cracker who can remove a new and very complicated protection scheme becomes a highly regarded and respected person within the cracker community.

How Crackers Crack: Debuggers and Disassemblers

Protection developers often presume that without source code, crackers will not be able to understand the software's protection. This is a huge mistake. Crackers use two kinds of utilities for breaking software protection—debuggers and disassemblers.

Debuggers

Debuggers allow crackers to trace an application, instruction by instruction, and to stop it at any point and follow its important sections. It is true that applications written in higher−level languages (like C++, Visual Basic, or Delphi) may be traced only in assembler, but crackers understand what is happening in the application code amazingly well—probably better than most people can imagine.

The truth is, the higher the level of the programming language, the more difficult it is to trace. But on the other hand, higher−level programming languages offer fewer possibilities for creating really good protection. Everything has its bright and dark sides.

Disassemblers

Disassemblers can translate application code back into assembler. One advantage that disassemblers offer over decompilers is that they always translate into assembler, so the cracker has to know only that one language. The quality of the resulting translated code depends on the quality of the disassembler. The best disassemblers even comment on the translated code, which makes the code that much easier to understand. For example, if the cracker finds a "Wrong serial number" string and locates its place in the code, he will be able to find the part of the code that protects the application. At that point, nothing can prevent him from studying the protection and breaking it.

Decompilers

Decompilers can translate application code back to source code. A decompiler can only translate applications that were written in the language for which the particular decompiler was created. There are, for example, decompilers for Delphi, Visual Basic, and Java. A good decompiler can do a good job of translating the application. Once an application is translated, it's easy for the cracker (if he knows the particular language) to find the sections of interest and determine how they work.

The Most Frequent Protection Failures

There are several reasons why a program may not be well protected against illegal copying:

- No program protection: It is very common for programs to contain no protection at all, and yet their authors require users to purchase the program. When a program is unprotected against copying, developers should not be surprised when their profits are small.
- Weak program protection: Approximately 70 percent of all programs have very weak protection, which crackers can remove very quickly.
- Program protection causing program failures: Many programmers protect their products weakly or not at all because they are afraid that incorrectly programmed protection will create problems with their programs.

It's better to use weaker protection code than none at all, but you will not stop the better crackers this way. Fine−tuning the protection scheme is the most important part of any protection strategy. Once the protection is created, the programmer should become a cracker for a while and, using the crackers' programs, test whether anything has been forgotten.

Chapter 2: Cracking Tools

Overview

If you don't know your enemy's weapons, you cannot defeat him. Let's take a look at the programs most commonly used by crackers.

SoftICE SoftICE from Compuware (http://www.compuware.com) is one of the best debuggers in the DOS environment. You will not find anything better for Windows 9x and NT. Many crackers therefore say that NuMega (the producer of SoftICE) is their favorite company. Since SoftICE is probably the best debugger, we will use it too, and we'll look at it more closely later in this chapter.

TRW2000 This is a debugger for Windows 9x. It isn't as good as SoftICE, but its price is acceptable considering the high quality. You'll find shareware versions online.

WinDasm Together with IDA (discussed below), WinDasm (shown in Figure 2.1) is the best disassembler in the Windows environment. Compared to IDA, WinDasm's disassembled code is shorter and easier to understand. It's a great loss that, unlike IDA, WinDasm is no longer in development. You can find shareware versions online.

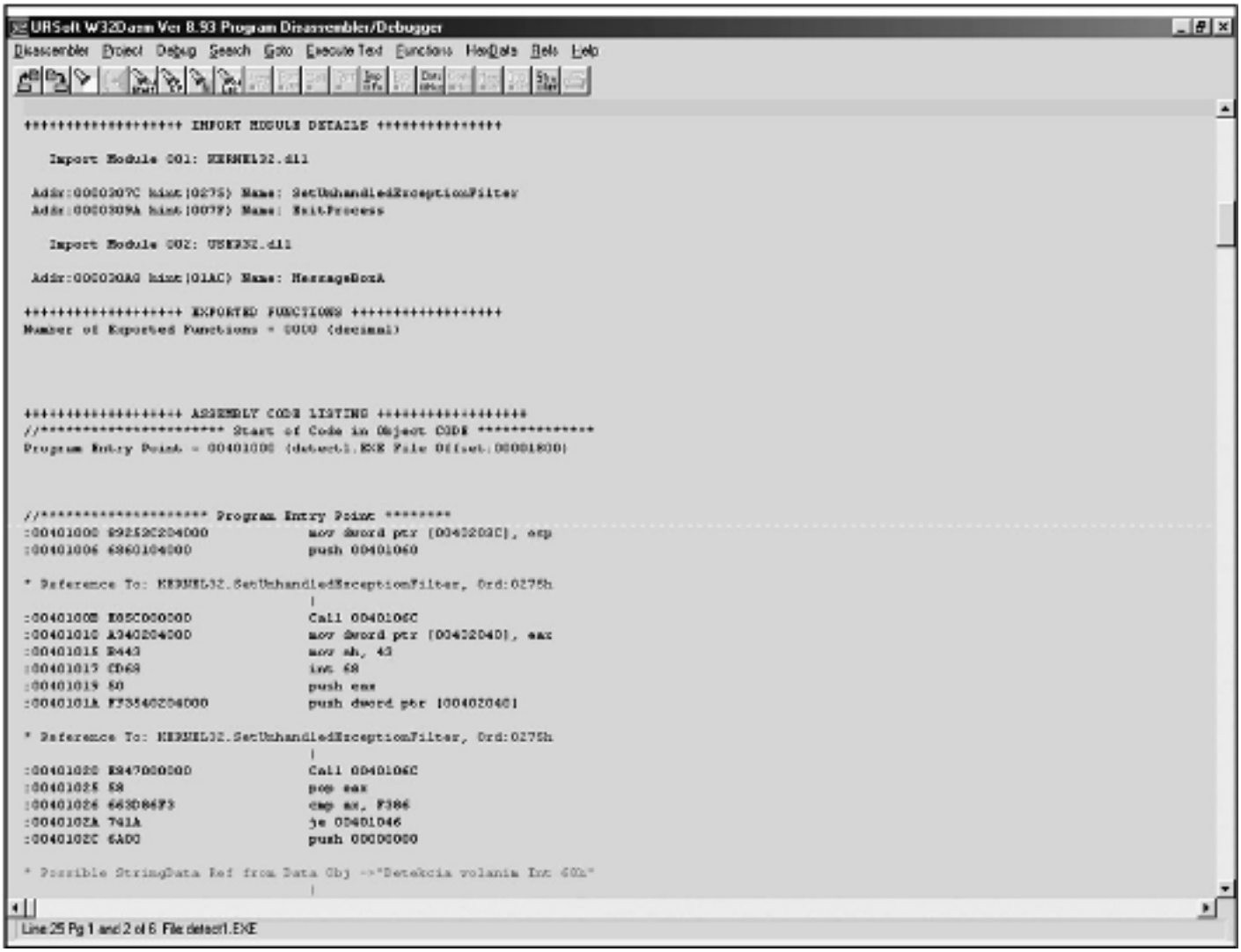

Figure 2.1: It is really easy to disassemble a program in WinDasm

SmartCheck SmartCheck from Compuware is an IDE tune−up tool for programs written in Visual Basic. It is better than SoftICE for debugging Visual Basic applications.

IDA Pro (Interactive DisAssembler Pro) IDA (shown in Figure 2.2), by Ilfak Guilfanov, is a wonderful disassembler for DOS and Windows programs. It is not a static disassembler like WinDasm, and it even lets you manage the translation manually. (This is a great feature to have when a program that you want to study uses various tricks to protect it from disassembly.) IDA has many other great features. You can request a demo of IDA Pro from http://www.ccso.com.

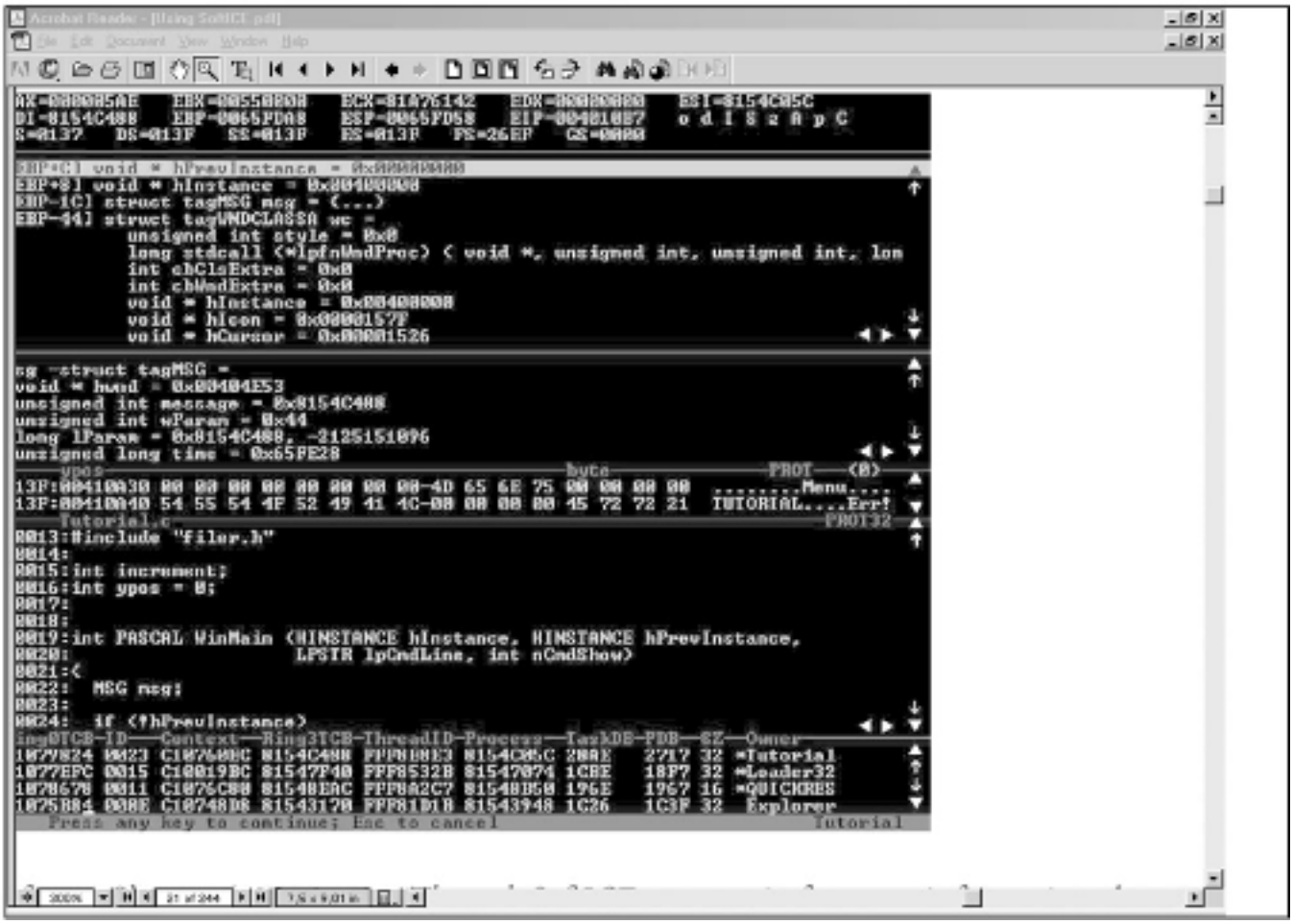

Figure 2.2: IDA looks like a DOS program, but it is a fully 32−bit application

Sourcer Sourcer, from VCOM, is a wonderful disassembler for DOS programs, but it is not widely used for Windows. You can get it at http://www.v−com.com.

Hex Workshop Hex Workshop, from BreakPoint Software (http://www.bpsoft.com) is a hex editor for the Windows environment.

Hiew (Hacker's View) Probably the best HEX editor for the DOS environment.

SoftICE Basics

As mentioned earlier, we will be using SoftICE in this book, so we'll take a closer look at it here. The SoftICE manual is an excellent and comprehensive resource (see Figure 2.3), so we'll just have a look at some of the basics of working with the program.

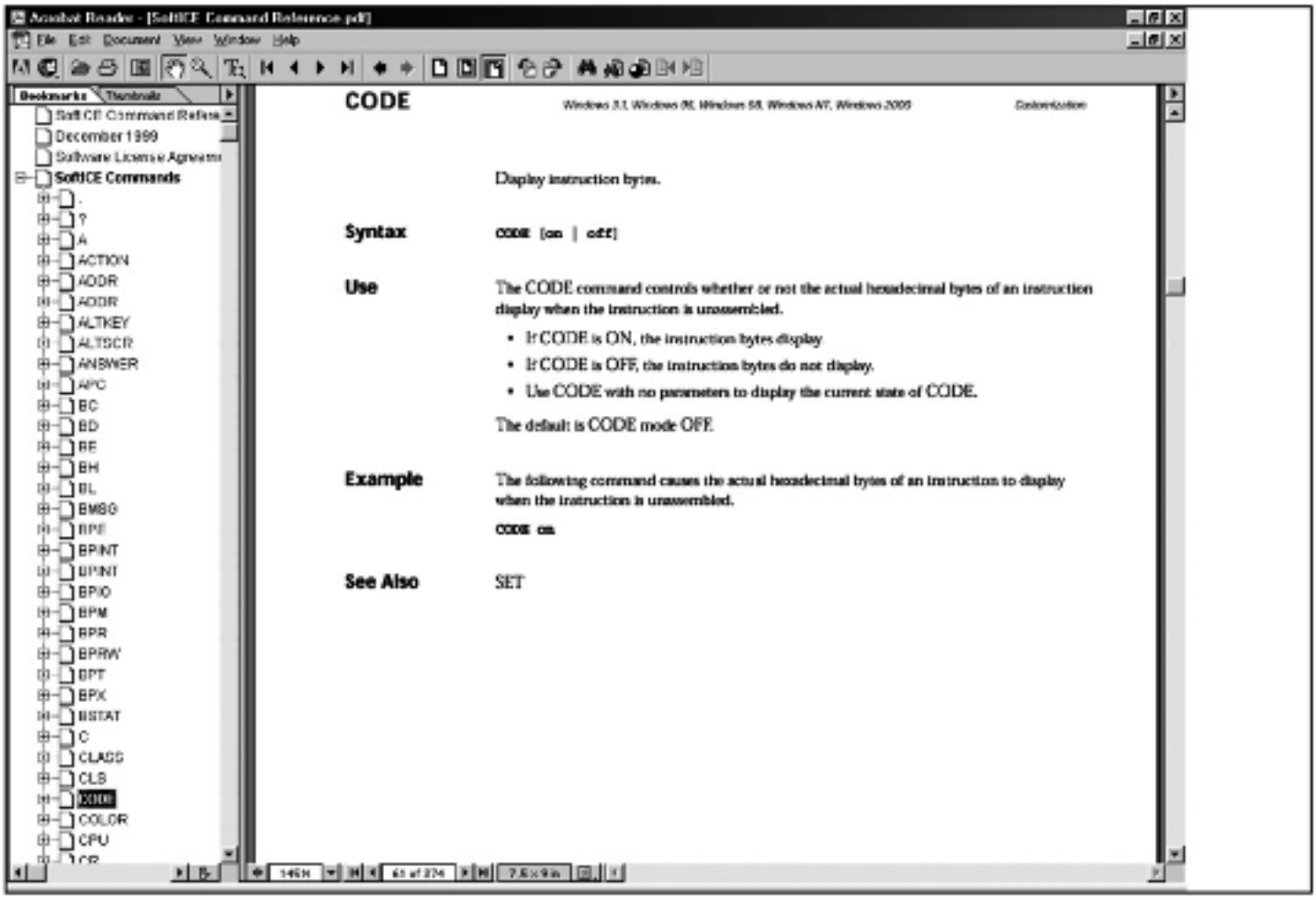

Figure 2.3: SoftICE contains wonderful and detailed documentation

Before you can work with SoftICE, you must enable Windows API calls. You can do so in SoftICE's winice.dat file where you will see the following text:

```
; *****Examples of export symbols that can be included for Windows 95*****
; Change the path to the appropriate drive and directory
```

You'll see various libraries listed below the preceding text, from which you can export symbols into SoftICE. Remove the semicolon (;) characters from in front of the kernel32.dll and user32.dll libraries. The text will then look like this:

```
EXP=c:\windows\system\kernel32.dll
EXP=c:\windows\system\user32.dll
```

You have just permitted functions to be exported to SoftICE from kernel32.dll and user32.dll and from their Windows API calls. Now you can set breakpoints for these calls in SoftICE. For example, you can directly use the command bpx MessageBoxA to set a breakpoint for this API call.

Another way to export to SoftICE is through the SoftICE loader menu, where you select Edit and SoftICE initialization settings. Select Exports in this menu and use the self−explanatory Add to add further exports and Remove to remove them.

Once you have made these changes, you must restart your computer so that SoftICE can be reinitialized.

In the following sections I will explain the basics of using SoftICE.

Key Commands

To get into SoftICE, you can use the key combination CTRL+D. This combination always works, whether you are at the Windows desktop or running a program or game. (Figure 2.4 shows what SoftICE looks like when it's running.)

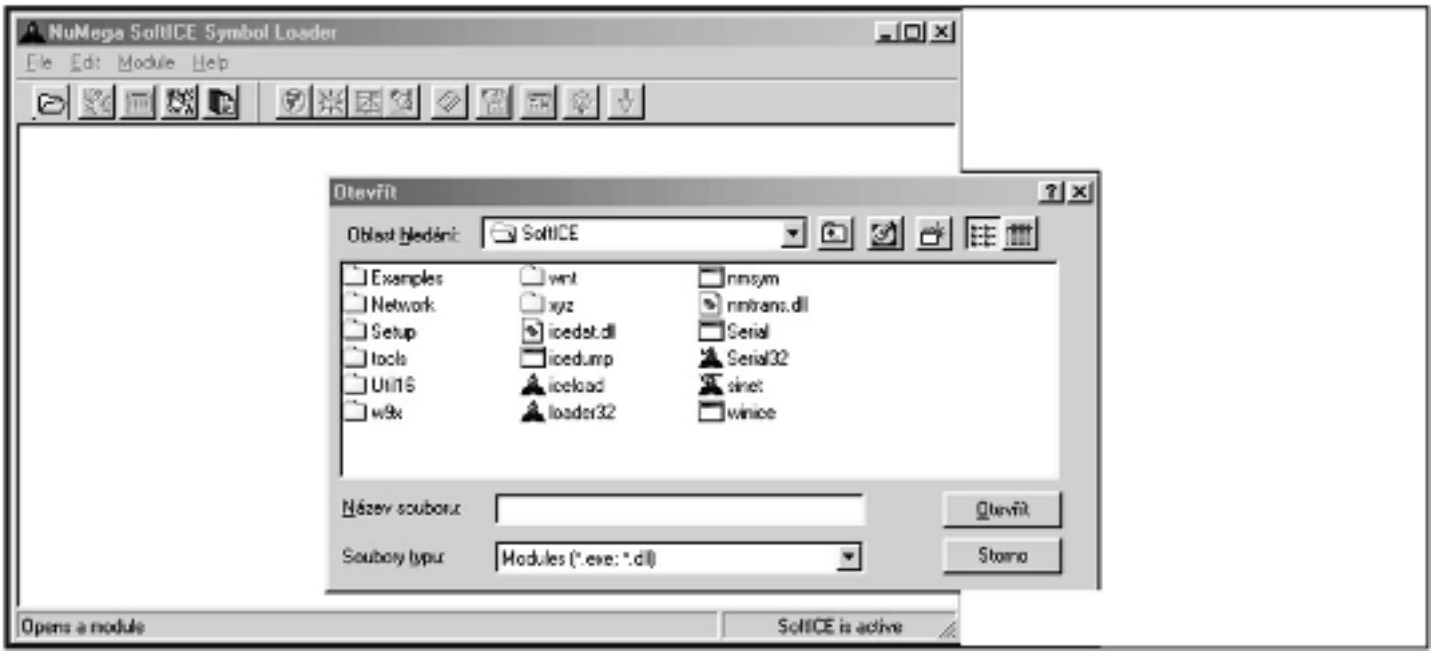

Figure 2.4: Running SoftICE during a program tune−up

If you press F10, the program you are debugging will be traced, one instruction after another, and the trace will not nest into call procedures. If you press F8 or entering the T (Trace) command, the program will be traced, one instruction after another, and the trace will nest into call procedures.

The F11 key is very important. If a breakpoint is set to an API call, SoftICE will stop at the beginning of this call. Pressing F11 again is like calling the RET function, though the API call will be performed before SoftICE stops. The advantage to this is that you don't have to perform manual call tracing, which can be time−consuming.

The BPX Command

The BPX [API call or an ADDRESS] command sets the breakpoint to that API call or address in the program. For example, BPX GETDRIVETYPEA would set the breakpoint to the Windows API GetDriveTypeA function. (You don't have to worry about lowercase or capital letters.) When using the BPX ADDRESS command, you enter a program address where the breakpoint should be set, and if the running program encounters this address, it will be stopped and you will be switched back into SoftICE.

The BPR Switch

The BPR [address1 address2] switch sets the breakpoint within a memory range, specified from address1 to address2. When anything reads from this range or writes to it, the program will be stopped and you will be switched into SoftICE. The switch has three options: r (read), w (write), and rw (read or write).

The BPM Switch

The BPM [address] command sets the breakpoint to a certain memory location. If anything reads from this location or writes to it, the program will be stopped and you will be switched into SoftICE. Like the BPR switch, this switch has three options: r (read), w (write), and rw (read or write).

If you use an x value as the switch, the so−called *debug breakpoint* will be set. This breakpoint will be written directly into the processor debug registers, and an INT 3h will not be set at the address, as with normal breakpoints. This kind of a breakpoint is much more difficult to discover.

Display Commands

The display commands are as follows:

- **d [address]** This command will show the memory contents in DWORD (4 bytes) beginning at the location defined by the address.
- **ed [address]** This command will let you edit memory contents in DWORD (4 bytes), beginning at the location defined by the address.
- **r [register value]** This command will change the register value. You can use it with conditional jumps.

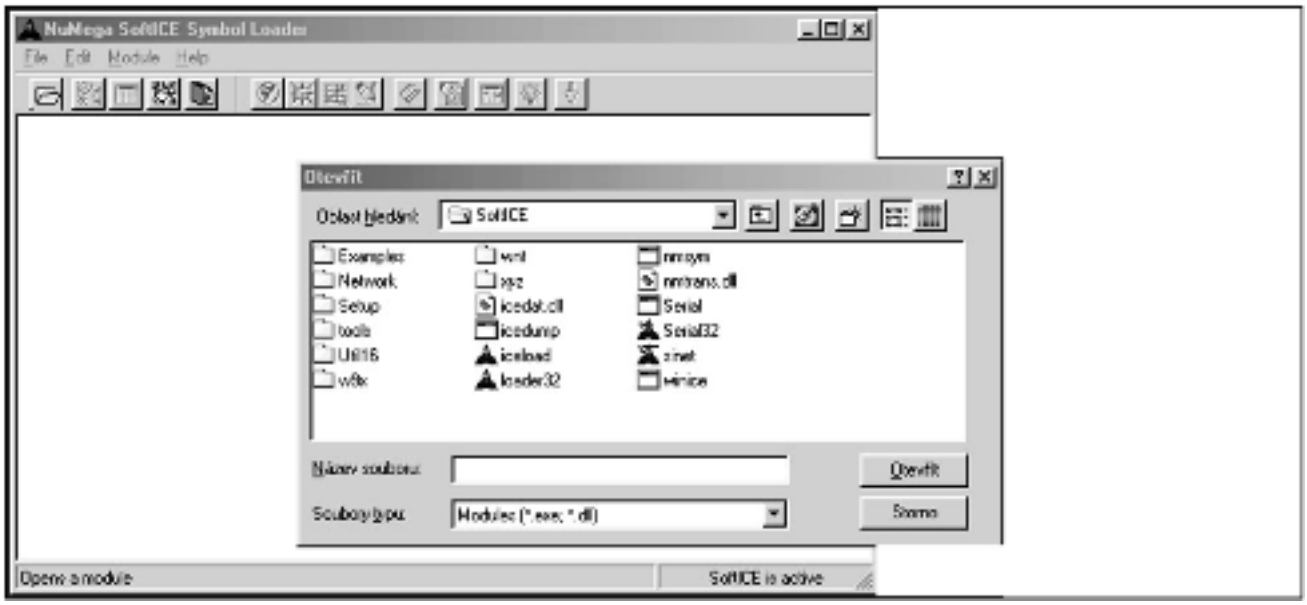

Figure 2.5: SoftICE Symbol Loader

You can also change special register values.

- **s [address1 address2 string or byte1, byte2 and so on]** This command will search the memory for a string or bytes from address1 to address2. For example, s 400000 401000 "test" will search for a "test" string from address 400000 to address 401000.
- **s** This command will continue searching for a string or bytes from the last found one.
- **code on** This command will show instruction prefixes.
- **wf** This command will show coprocessor register values.
- **exp** This command will show exports.
- **rs** This command will show the program window in the actual state, and will return to SoftICE when you press any key.
- **address** This command will let you insert program code in assembler directly from the entered address.
- **hboot** This command will reset the computer. It is useful in case of a system crash.

Of course, SoftICE also contains many other commands. You can find all of them in the SoftICE documentation.

Chapter 3: The Basic Types of Software Protection

In this chapter I will describe most types of contemporary software−protection programs, all of which have their pros and cons. Which is best depends only on your opinion of the program code and the creators' preferences.

Registration−Number (Serial−Number) Protection

Programs that use registration−number protection require the user to enter a registration number to register the program. The registration number depends on specific criteria.

Programmers use different types of registration−number protection, including the following:

- Registration number is always the same.
- Registration number changes in accordance with entered information (company, name, and so on).
- Registration number changes in accordance with the user's computer.
- Registration−number protection in Visual Basic or Delphi programs.
- Registration number is checked online.

Registration Number Is Always the Same

A program protected with this method requires the user to enter a registration number (see Figure 3.1). However, because the registration number is always the same, the cracker only has to find the correct registration number, post it online, and the program can then be registered by anyone.

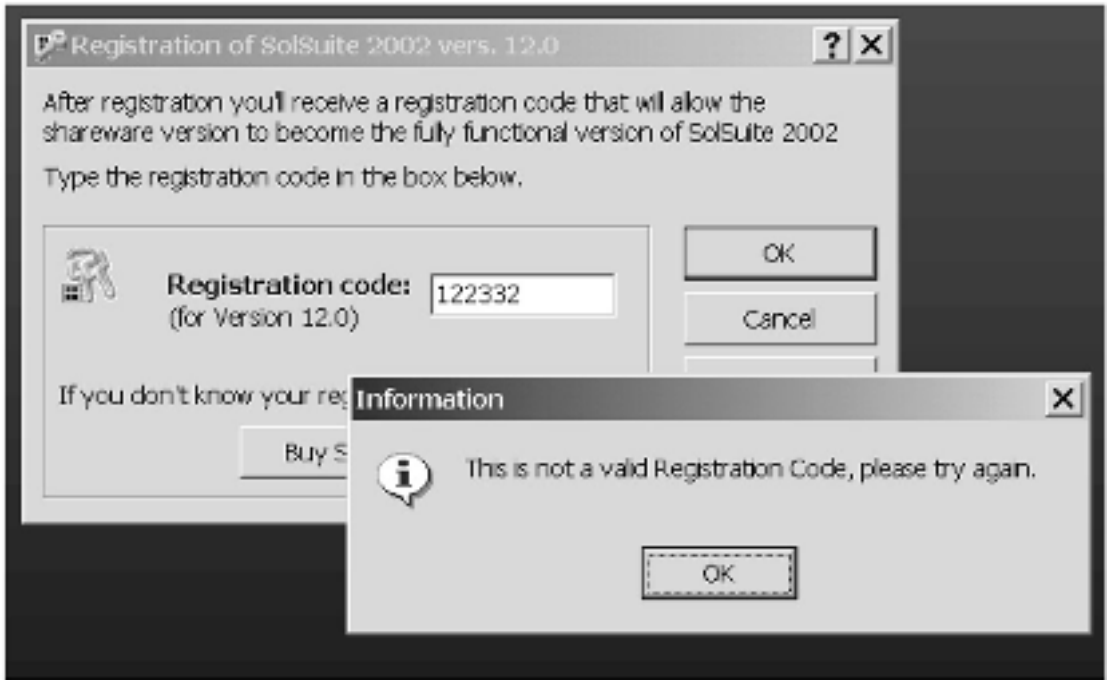

Figure 3.1: Registration number is always the same

One advantage of this method of protection, when compared with other registration−number protection techniques, is that the correct registration number doesn't have to be saved in memory to be compared with the entered number, which will often be XORed or recalculated in some other way. The correct registration number will then also be recalculated and both results compared. Naturally you can use more complicated calculations that are not easy for the cracker to understand, making it difficult to get from the result back to the correct registration number.

You can make excellent use of this protection method by encoding several program sections, such as a Save function, with the correct registration number value. If a cracker uses the patch method (directly adjusting conditions in the program code) and the correct registration number hasn't been entered, the

originally blocked functions will still not work correctly.

It isn't a good idea to decode the blocked sections right after the correct registration number has been entered. It is safer to decode these sections only when the program has been started, or better still, only after the blocked function has been called. If the function is encoded again after use, and the program contains many encoded sections, the program will never be decoded in the memory as a whole, which means that a dump from memory will not help the cracker very much.

This software protection should be combined with other types that will be described later on.

Registration Number Changes in Accordance with Entered Information

This is the most frequently used type of protection. In this case, before you enter the registration number, you have to enter a user name, company, or other information, and the correct registration number changes according to the information you enter (see Figure 3.2). If you enter a registration number that doesn't match the information entered, the registration won't succeed (see Figure 3.3). The more skilled the programmer, the more difficult he can make it for the cracker to break this protection. However, even though the calculation algorithm may be very complex, once the user has entered the registration number, it is compared with the calculated one, and the cracker only has to trace the program to find the correct registration number.

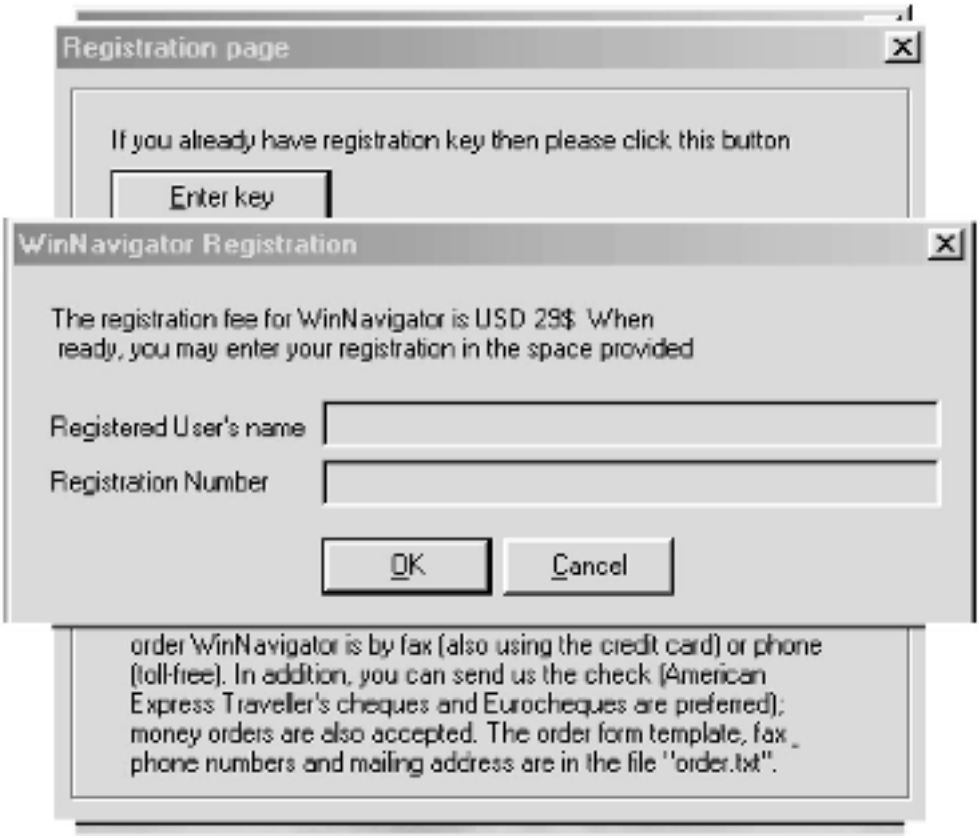

Figure 3.2: Registration number changes in accordance with the entered name

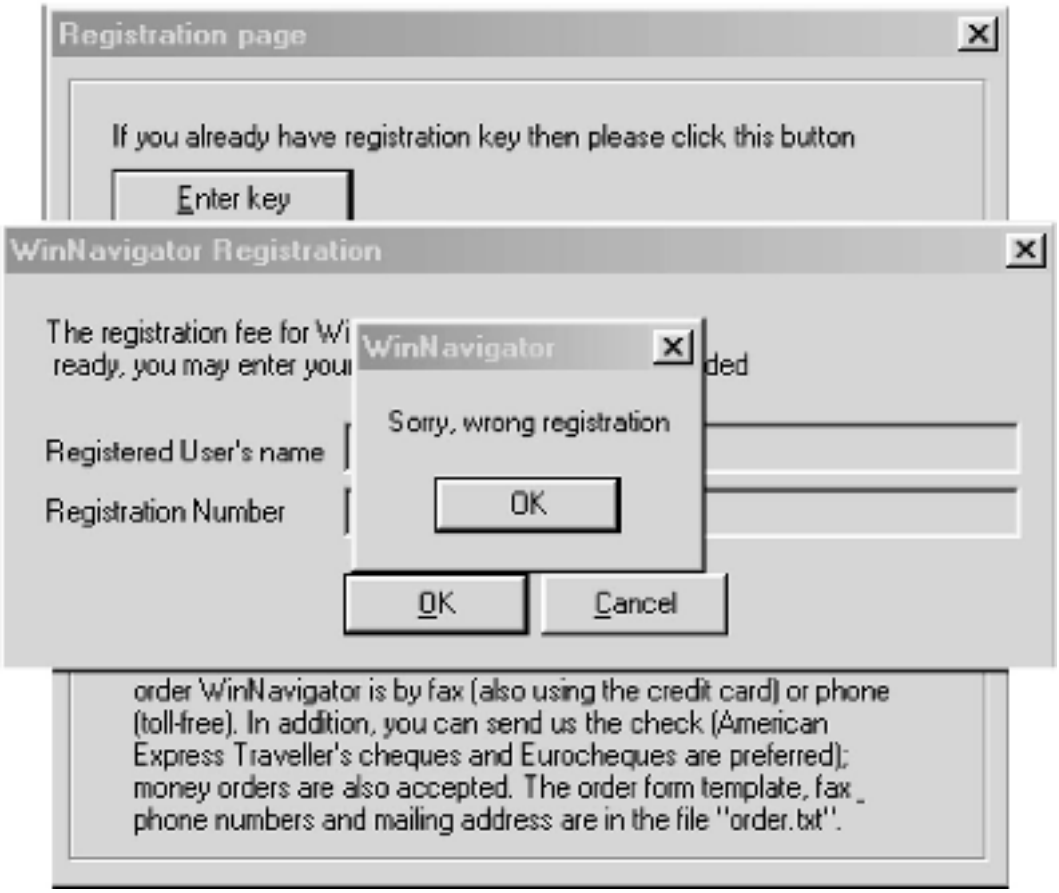

Figure 3.3: When an incorrect registration number is entered, the registration is unsuccessful

To buttress this protection, it is a good idea to design the algorithm so that the entered name and registration number must produce a certain result. Most programs don't institute this control—the registration number doesn't have to be checked, and it can offer a range of results. The attacker may exploit this range and generate a registration number in accordance with his own wishes.

When this sort of protection is used, the program should contain another hidden algorithm to check whether the entered registration number was really the correct one. It shouldn't tell the user, though, if it finds any inconsistencies. It will be enough if the user is somehow punished. (I will leave this up to the programmer's imagination.)

You can also encode some parts of the program (as mentioned previously) so that they cannot be used in an unregistered version. Then, once you've verified part of the registration number, you can use the unverified part for decoding the parts of the program. If you use good scrambling and a sufficiently long code, it will be almost impossible for the cracker to find the correct value for decoding. ASProtect (described in Chapter 6) works in this way.

Registration Number Changes in Accordance with the User's Computer

This is an unpleasant type of protection for an attacker, and it may even fool an inattentive cracker because, although he will register the program at his computer, it will not be possible to register the pirate version anywhere else. The registration number may change, for example, in response to the hard drive serial number or according to some random sequence. (It is important to hide this registration number carefully, because if it is found, it could easily be changed into a uniform number and the program could be registered at any computer with the same registration number.) Naturally, the registration number should also change in accordance with other data (such as user name, company, and so on) so that it works for only one user (see Figure 3.4).

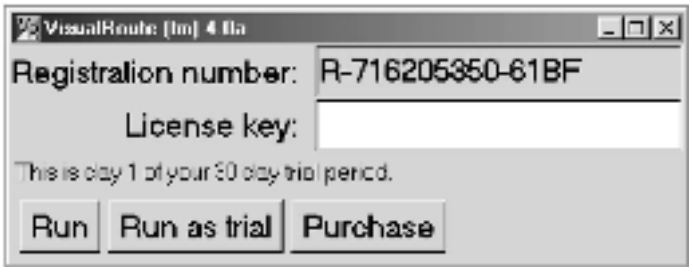

Figure 3.4: Every computer requires a different registration number
When using this type of protection, it is very important to hide both the code that detects the computer number and the checking code that assesses the correctness of the entered registration number. It is good to combine this method with other types of protection.

Registration−Number Protection in Visual Basic Programs

Protecting a program in Visual Basic (VB) isn't easy, since the programming language itself is a high−level language (the higher the language level, the further the compilation is from assembler). With high−level languages it is difficult to influence the compiled source code.

VB programs fall into the following groups:

- VB4
- VB5 and higher
- VB5 and higher, compiled in p−code

We'll look at each of these in turn.

VB4

Although it may not be obvious to most users, the VB4 family of programs has very inadequate protection, and the experienced cracker will find most registration numbers within five minutes (see Figure 3.5). The trouble is that VB4 programs mostly use the same VB40016.dll (VB40032.dll) library call for comparing the entered registration number to the correct registration number.

Figure 3.5: At first sight, a registration requirement programmed in Visual Basic isn't different from other languages

Even when a VB4 program uses some other comparison method, it is usually possible to find the correct registration number easily in memory. The registration number is usually placed close to the entered number, so all you need to do is search the memory for the entered registration number to find the correct one.

One advantage of VB4 programs is that they are hard to trace in SoftICE because their code isn't compiled into a machine code but only into pseudoinstructions that will be performed after the program is started. There are decompilers for VB4, but they are rarely used.

How VB4 Programs Are Cracked

How are VB4 programs cracked then? Even though VB4 is only rarely used for programming these days, it is good to know about it to avoid the mistakes in its use.

For a 16−bit VB4 program, the cracker has to find something like the following in memory and in the VB40016.dll code:

```
8BF88EC21EC5760E33C0F3A674051BC01DFFFF
```

He then sets a breakpoint to the address where the code was found, and he is able to see the following program instructions in the debugger:

Prefix	Instruction
.	
.	

.	
8BF8	mov di,ax
8EC2	mov es,dx
1E	push ds
C5760E	lds si,[bp+0E]
33C0	xor ax,ax
F3A6	repz cmpsb
7405	jz 2667
1BC0	sbb ax,ax
1DFFFF	sbb ax,ffff
.	
.	
.	

Now all the cracker needs to do is to see what he finds at the address where the repz cmpsb instruction compares the entered registration with the correct one. (You have only to look at the addresses es:di and ds:si to find the correct one.) This very simple tactic usually works, which probably isn't good news for most 16−bit VB4 programmers. Fortunately, 16−bit VB4 is used less and less. When it is, I recommend using a good DOS compressor or some other EXE protector for the program itself.

Most 32−bit VB4 programs use the MultiByteToWideChar function in the VB40032.dll library to compare two strings. During registration, all the cracker needs to do is set a breakpoint to the hmemcpy function before clicking OK, and then trace the program until he gets into VB40032.dll and finds the following bytes in the memory:

```
56578B7C24108B74240C8B4C2414
```

Then, he sets a breakpoint to the address found, and after repeatedly clicking OK, he will see the following code:

Prefix	Instruction	Explanation
.		
.		
.		
56	push esi	
57	push edi	
8B7C2410	mov edi, [esp + 10]	;es:edi −−> registration number should be here
8B74240C	mov esi, [esp + 0C]	;esi −−> correct registration number
813F70006300	cmp dword ptr [edi], 00630070	
7527	jne 0F79B381	
803E00	cmp byte ptr [esi], 00	
.		
.		
.		

VB5 and Higher

The software protection in VB5 and higher is slightly more difficult for crackers to tackle than that found in VB4. Many crackers are rather disappointed when they find out that they have a VB5 program in their hands, and they often leave it alone, but the experienced ones aren't scared off that easily.

Crackers aren't keen on cracking VB programs (or those written in Delphi) because the code is hard to read and understand, and it takes a lot of time to trace. To crack VB5, they usually use methods for generating a registration number for just one user, or modifying the program code to allow for any registration number. Only rarely does someone create a registrationnumber generator for a VB5 program, and then only the best crackers are able to do so because the code is so difficult to understand. (They may use SmartCheck, since it helps them understand the code.)

The fact that VB5 programs are so unpopular among crackers, and that it's difficult to create registration−number generators for them, is one advantage of using VB5. When a good compression program or protector is used, VB5 programs can be a challenge for any cracker.

It is a good idea to use a registration number that changes on different computers. This involves heavy interference with the program code, but the result is that it is almost impossible to use anti−debugging tricks.

VB5 and Higher, Compiled in P−Code

Programs compiled in *p−code* (short for packed code) are probably the VB programs least liked by crackers. Unlike VB4 programs, p−code programs are translated into pseudo−instructions, not machine code, and these instructions are performed when the program runs. Programs in p−code can be traced in SmartCheck, but not easily. The most experienced crackers will try to trace such programs in SoftICE, but only a small group of them will be able to do so.

VB programs compiled in p−code combine well with other software protection, and when they are used with a good compression program or protector, they are a challenge for any cracker. As a programmer, you should use registration numbers that change with different computers. Doing so will heavily interfere with the program code, but it will make it nearly impossible to use anti−debugging tricks.

Registration Number Is Checked Online

Some newer programs use the latest in modern technology for testing the correctness of a registration number (see Figure 3.6). Once the registration number is entered, the program sends it via the Internet for verification. A server then tests the number and returns a report which tells the program whether the number is correct (see Figure 3.7). The program processes this report to determine whether the program was properly registered. Despite this online exchange, though, most of these protection programs are very simple, and an experienced cracker can get rid of them quickly and reliably.

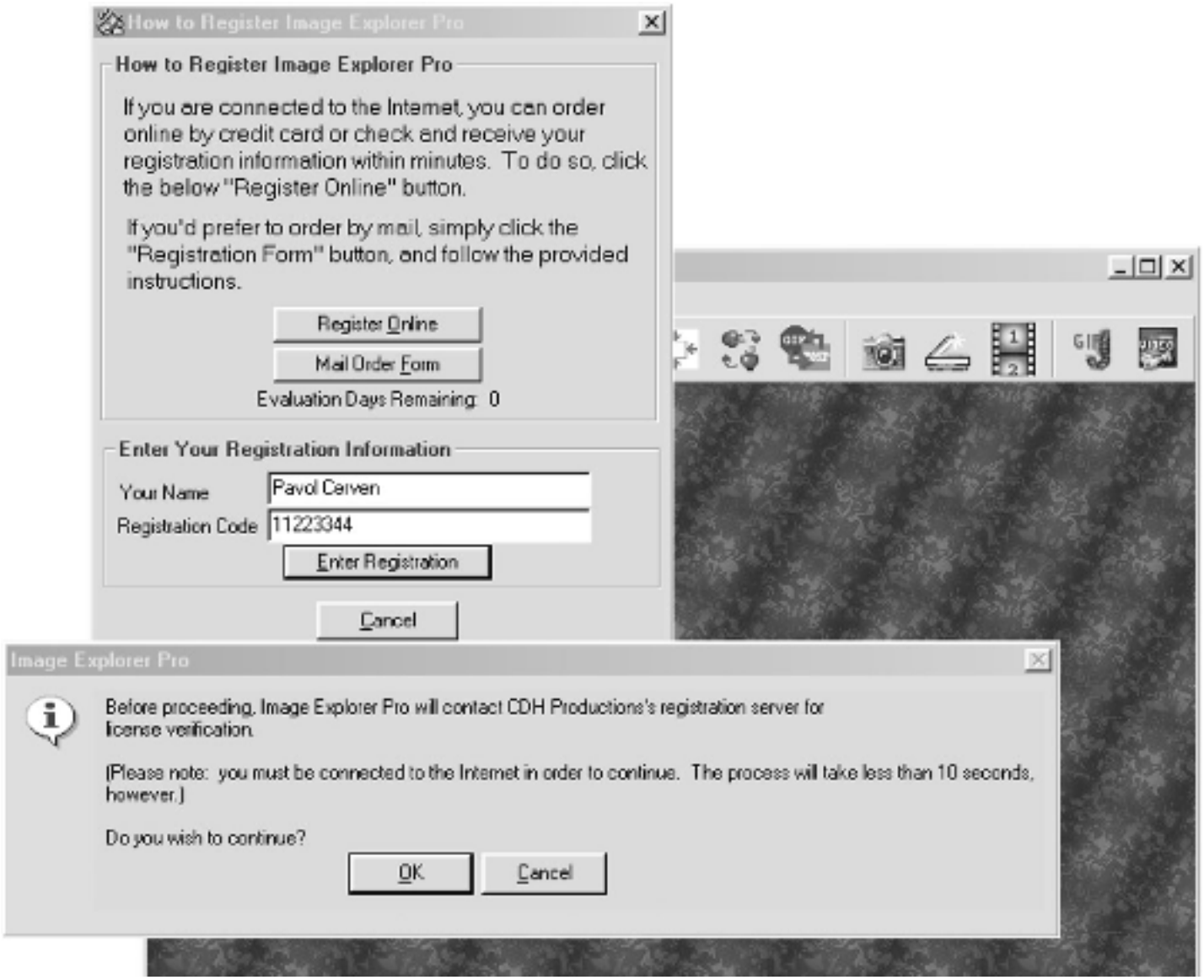

Figure 3.6: A program is ready to send registration information to the control server via the Internet