5

PROGRAM COMPRESSION AND ENCODING: FREEWARE AND SHAREWARE

Compressing or encoding programs is an excellent way to add additional product protection. One advantage of compressing a program is that the program must be uncompressed before it can be changed. Also, better compression programs complicate program debugging, and, of course, you cannot disassemble a compressed file. (While it may still be possible to create a loader that will change the program code directly in memory once the program has been decompressed, no well-protected program should allow anything like this.)

Compressing all of a program's files makes it difficult to change things like text, but compressed programs will run a bit more slowly than uncompressed ones (though the performance hit with a good compression program will be insignificant.) Rather than compress all of a program, you can compress executables only—this will not slow the program down at all, though start-up will be slightly slower (because compressed programs are decompressed in memory during start-up).

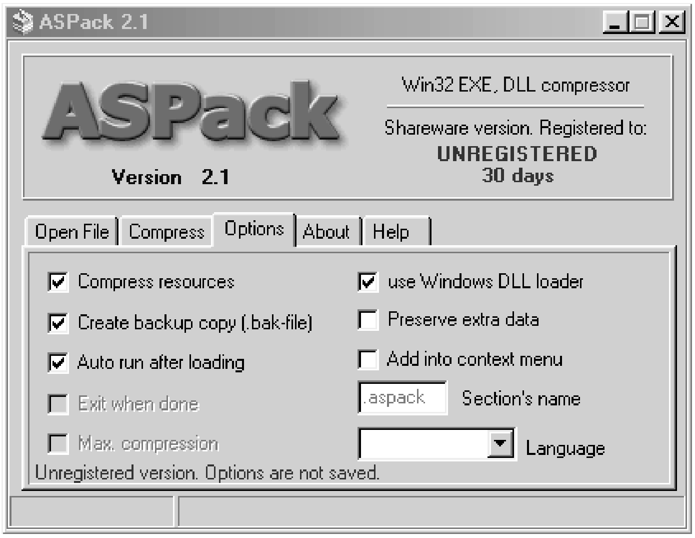
When deciding whether to use compression, find out whether there is a decompressor for the particular compressor you want to use. If so, don't use it. For example, while PKLITE is the best compressor for EXE files in DOS, there are many decompressors. Your best bet, of course, is to create a new compressor. Still, most programmers will use the preprogrammed ones, so we'll take a look at a variety of compression and encoding programs for executables, as well as some other types of commercial protection.

aPLib

aPLib (home19.inet.tele.dk/jibz/apack/index.html) is a commercial compression library for programmers who want to compress data in their programs, created by the great programmer Joergen Ibsen. aPLib is used by many compression programs for executable files because it is one of the best products in the field.

ASPack

ASPack (www.aspack.com) is a compression program for EXE, DLL, and OCX files. It is easy to use, even for less-experienced users. On the other hand, more-experienced programmers may not like it because the program menu doesn't offer many options, as shown in Figure 5.1.



☑ Compress resources
☑ Create backup copy (.bak-file)
☑ Auto run after loading
☐ Exit when done
☐ Max. compression
☑ use Windows DLL loader
☐ Preserve extra data
☐ Add into context menu
.aspack Section's name
Language
Unregistered version. Options are not saved.

Figure 5.1: You cannot select maximum compression in the unregistered version of ASPack

ASPack's decompression routines contain several jumps that are designed to confuse a cracker but which really present problems only for the inexperienced ones. For example, ASPack's decoding routine contains only a few anti-debugging tricks. If a file is loaded into SoftICE's Symbol Loader, the program will not stop at the beginning. However, once you find the beginning of the program, insert INT 3h or an equivalent there, and set the breakpoint to this INT, then run the program, it will stop right at the beginning. Or, you can load the program into ProcDump and change the characteristics for the .text section. Here you will probably find C0000040; change that to E0000020 and the program should always stop at the beginning in the SoftICE loader. (I'll describe this more fully in Chapter 7.)

To remove ASPack, all you need is a special decompressor, though you can also create a loader that will change the program code directly in memory. (ASPack has no protection against anything like that.) I have even seen a program on the Internet that makes it possible to create these patches for ASPack.

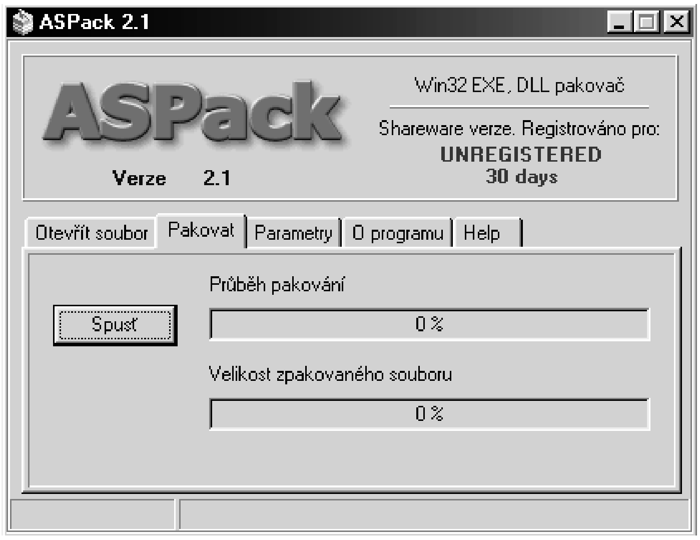


Figure 5.2: ASPack lets you choose from several languages

ASPack's Option menu contains the following items:

Compress Resources Compresses resources along with the program.

Create Backup Copy (.bak-file) Creates a copy of the compressed file.

Auto Run After Loading Compresses the program automatically as soon as it is loaded.

Exit When Done Closes ASPack once a program has been compressed.

Max Compression Compresses the program as much as possible.

Use Windows DLL Loader Uses the Windows DLL loader (this is important if you have an old Borland C++ linker).

Preserve Extra Data Preserves extra data located after the end of a file. (This mostly concerns files with overlay data.)

Add Into Context Menu Adds an ASPack item into the Explorer menu. If you right-click on a file, you can immediately compress it with ASPack.

Section's Name Specifies a section name for the decompressor data in the compressed file (this is a kind of author's mark).

Language Sets ASPack's language.

ASPack is a very good compression program, but it needs to be toughened up with more anti-debugging tricks. On the other hand, because its decompression routine doesn't contain any incompatible operations, it shouldn't cause problems with most programs, and it does offer very good compression.

In recent versions, the ASPack programmers have focused on dumping the program from memory in an effort to protect the import table as much as possible. They seem to have forgotten, though, that without good anti-debugging tricks and anti-disassembling macros, it is very easy to trace and view the ASPack code, so it will not take long before a new decompressor appears.

Test file compression: 486,400 bytes (using an unregistered version of ASPack with maximum compression option turned off)

Decompressors: ProcDump and UnASPack

Ding Boys PE-Crypt

Ding Boys PE-Crypt (shown in Figure 5.3) is another commonly used executable file encoder. It's particularly interesting because it implements anti-debugging tricks designed to make it impossible to run an encoded program when a debugger is present in memory. The creator's radical solution to this problem is also interesting because the program will freeze without warning if a debugger is in memory. Still, the program's encoding isn't too difficult, and it's no surprise that there is a decoding program freely available on the Internet.

The decoding routine in Ding Boy's PE-Crypt is clearly intended to make manual decoding annoying and time consuming. For one thing, it uses loops that only ever decode the following loop. Also, every loop decodes only 29 bytes, and each contains anti-debugging code, which means that the cracker has to remove this code from each individual loop. Manual decoding would therefore take several hours. (Of course, if you use a decoder, it will take only a few seconds.)

Once you run the program, you will see a menu from which you can select several encoding functions, as discussed in the list below. At the top you'll see a field where you can type a path to the program that you want to

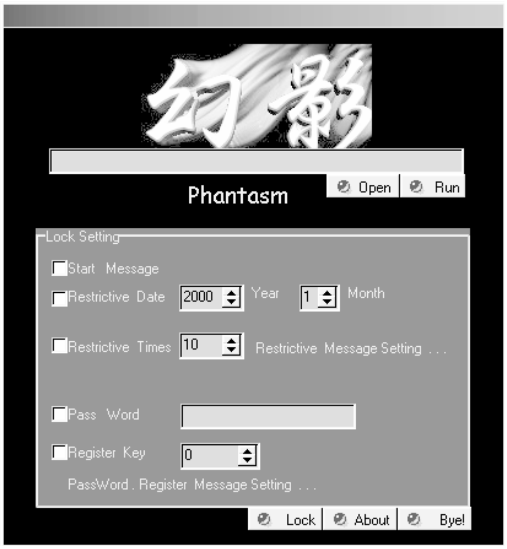

Phantasm
Open
Run
Lock Setting
Start Message
Restrictive Date 2000 Year 1 Month
Restrictive Times 10 Restrictive Message Setting . . .
Pass Word
Register Key 0
PassWord . Register Message Setting . . .
Lock
About
Bye!

Figure 5.3: Luckily Ding Boy's PE-Crypt has a menu in English

encode, or you can click Open and browse to the program. Use Run to start the program.

Start Message Sets a message that will appear when the program starts.

Restrictive Date Sets a date on which the program will expire. (It is surprising that you can set only months.)

Restrictive Times Sets how many times the program can be run.

Restrictive Message Setting Specifies a message to appear after one of the time limits expires.

Pass Word Sets a password that will always be required to run the program.

Register Key Sets a registration number for the program. (Ding Boy's PE-Crypt supplies you with a program that will calculate such a number for you.)

Password. Register Message Setting Sets the text of a message that will appear after an incorrect password or registration number has been entered.

Ding Boy's PE-Crypt is definitely one of the better encoding programs and, because it is not too widely used, it may be more effective against crackers. Manual decoding of PE-Crypt-encoded files is both difficult and very time consuming, though who knows whether decoders will really have a problem removing Ding Boy's PE-Crypt.

7

ANTI-DEBUGGING, ANTI-DISASSEMBLING, AND OTHER TRICKS FOR PROTECTING AGAINST SOFTICE AND TRW

Few programmers realize just how important it is to prevent their code from debugging or disassembly. It's essential to do so, because if a program can be debugged, it is easy for a cracker to understand how the protection works. Fortunately, even the simplest anti-debugging tricks can complicate debugging, and we can use anti-disassembling macros to make it harder to understand the debugged code. When well-executed, both tricks will make it much more difficult for a cracker to remove even the simplest protection.

We've had a look at the compression and encoding programs that many programmers rely on to do the dirty work of software protection. Using these programs alone, though, is really not a good solution, because it will only be a matter of time before a new decoder becomes available, and before the better crackers remove the encoding programs themselves, thus leaving the application without protection against debugging.

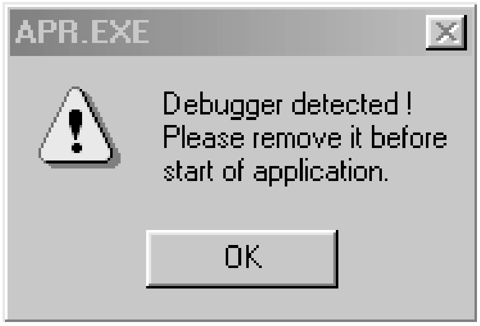
Anti-debugging and anti-disassembling tricks aren't hard to use, but it is very important to test your protected application carefully.

NOTE *As of this writing, there are only a few anti-debugging tricks for Windows NT, Windows 2000, and Windows XP because their internal structure differs from that of Windows 9x. You may find it necessary to have your program test to see which operating system is running and then decide which trick to use. Anti-disassembling tricks, on the other hand, are not operating-system-dependent and will work without such problems. It is therefore advisable to use them as much as possible.*

All of the example programs discussed in the following pages are written in assembler, and it is a good idea to avoid programming them in a higher-level programming language. While not every programmer knows assembler, most higher-level languages let you insert assembly code. If your language of choice does not, it will be much more difficult for you to work with the code you'll find here. In that case, your best bet will be to insert the anti-debugging tricks into a separate DLL library, and then to call those functions from the protected application. (Of course, this is not ideal, because a cracker will be able to remove such protection quickly and easily.)

NOTE *You only need to use anti-disassembling macros to protect critical points in the program that should remain hidden from an attacker. You do not need to use these macros throughout the program.*

Your program should perform a simple test for the presence of a debugger in memory as soon as it starts. If a debugger is found, the program should refuse to run and possibly display a warning that the debugger should be removed (see Figure 7.1). While a cracker will probably be able to get around the first test easily, you should consider having the program, in a later test, check a second time for the presence of a debugger. Then, if it finds that a debugger is still active, the program could "freeze," or do something else to make it difficult or impossible for the cracker to continue. I do not, however,


APR.EXE

Debugger detected !
Please remove it before
start of application.

Figure 7.1: ASProtect has just found an active debugger

recommend displaying a warning under any circumstances, because such a warning makes it clear that this is an attack.

The examples that follow are tuned for Turbo Assembler v5.0 from Borland. The code isn't completely optimized and should therefore be understandable by less-than-expert assembler programmers. (I hope that the pros will excuse me.) While it wouldn't be hard to optimize the examples, doing so would result in less-readable code.

My advice to those of you who don't know how to work in assembler is to learn it. Even though many people claim that it is a dead language, there are still fields where it is necessary, and programming good software protection is one of them, especially when tuning your application (which can only be done well in assembler). When tuning your code, you will discover the holes that make it easy for crackers to conquer and remove software protection.

Let's begin.

Detecting SoftICE by Calling INT 68h

Here's a way to detect the presence of SoftICE in memory by calling `INT 68h`. The AH register must contain the value 43h before calling `INT 68h`. If SoftICE is active in memory, the return value 0F386h will be in the AX register.

This is a well-known method of detecting SoftICE that is safe and commonly used, but only in Windows 9x. You can see it in action, for example, in SafeDisc:

1

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
include w32.inc
Extrn SetUnhandledExceptionFilter : PROC
.data
message1    db "Detection by calling INT 68h",0
message3    db "SoftICE found",0
message2    db "SoftICE not found",0
delayESP    dd 0                                ;the ESP register saves here
previous    dd 0                                ;the ESP register will save the address of the
                                                ;previous SEH service here

.code
Start:

;---------------------------------------------------------------------------------------------
;Sets SEH in case of an error
;---------------------------------------------------------------------------------------------
```

```
            mov  [delayESP],esp
            push offset error
            call SetUnhandledExceptionFilter
            mov  [previous], eax
;-------------------------------------------------------------------------------------------------
;The new address for Structured Exception Handling (SEH) is set here to ensure that in case of an
;error, the program will continue from an error label and will end correctly. This is important
;if, for example, the program calls an interrupt that will be performed correctly only if SoftICE
;is active, but which will cause an error and crash the program if SoftICE is not active. Finally,
;the previous SEH service address is saved.
;-------------------------------------------------------------------------------------------------
            mov  ah,43h                        ;service number
            int  68h                           ;calls the INT 68h interruption
            push eax                           ;saves the return value
;-------------------------------------------------------------------------------------------------
;Sets previous SEH service
;-------------------------------------------------------------------------------------------------
            push dword ptr [previous]
            call SetUnhandledExceptionFilter
;-------------------------------------------------------------------------------------------------
;Sets the original SEH service address
;-------------------------------------------------------------------------------------------------
            pop  eax                           ;restores the return value
            cmp  ax, 0f386h                    ;tests to see whether the return value is
                                               ;a "magic number"
;-------------------------------------------------------------------------------------------------
;If SoftICE is active in memory, the return value will be F386h in the AX register.
;-------------------------------------------------------------------------------------------------
            jz   jump                          ;if yes, the program jumps because SoftICE is
                                               ;active in memory

continue:
            call MessageBoxA,0, offset message2,\
            offset message1,0
                                               ;if the return value was other than F386h,
                                               ;SoftICE was not found, and an error message
                                               ;will be displayed.
            call ExitProcess, -1
                                               ;ends the program

jump:
            call MessageBoxA,0, offset message3,\
            offset message1,0
                                               ;prints a message that SoftICE was found. Any
                                               ;code may follow from this point.
```

```
call ExitProcess, -1
                                                    ;ends the program
error:
                                                    ;starts a new SEH service in case of an error.
            mov  esp, [delayESP]
            push offset continue
            ret
                                                    ;if an error occurs in the program, SEH
                                                    ;ensures that the program will continue from the
                                                    ;error label.


ends
end Start
                                                    ;end of program
```

Detecting SoftICE by Calling INT 3h

This is one of the most well known anti-debugging tricks, and it uses a back door in SoftICE itself. It works in all versions of Windows, and it is based on calling INT 3h with registers containing the following values: EAX=04h and EBP=4243484Bh. This is actually the "BCHK" string. If SoftICE is active in memory, the EAX register will contain a value other than 4.

This trick has often been used in the code of various compression and encoding programs, and it is well known because of its wide use. When used well, it may cause trouble even for the more experienced crackers.

2

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
include w32.inc
Extrn SetUnhandledExceptionFilter : PROC
.data
message1    db "Detection by calling INT 3h",0
message3    db "SoftICE found",0
message2    db "SoftICE not found",0
delayESP    dd 0                                    ;the ESP register is saved here.
previous    dd 0                                    ;the ESP register will save the address of the
                                                    ;previous SEH service here.

.code
Start:
```