Software Protection, an impossible dream?

All the advices you may need

1 Never use meaningful file or procedure names such as IsValidSerialNum (duh.) If you do use functions for checking purposes, place at least some required code that your

program really needs, in such a function. When the cracker disables the function, the program will produce incorrect results.
2 Don't warn the user right after a violation is made. Wait later, maybe until the next day or two (crackers hate that).
3 Use checksums in DLL's and in the EXE. Have them check each other. Not perfect but it just makes it harder to crack.
4 Pause a second or two after a password entry to make brute force cracking unfeasible. Simple to do, but rarely done.
5 Self-heal your software. You know, error correction like modems and hard drives use. The technology has been around for years, and no one uses it on their software?

The best thing about this is that if the cracker used a decompiler, they may be looking at a listing that is no longer valid.
6 Patch your own software. Change your code to call different validation routines each time. Beat us at our own game.
7 Store serial numbers in unlikely places, like as a property of a database field.
8 Store serial numbers in several places
9 Don't rely on the system date. Get the date of several files, like SYSTEM.DAT, SYSTEM,DA0 and BOOTLOG.TXT and compare them to the system date. Require that the

time be greater than the last run.
A Don't use literal strings that tell the user that their time is expired. These are the first things to look for. Build strings dynamically or use encryption.
B Flood the cracker with bogus calls and hard-coded strings. Decoys are fun.
C Don't use a validation function. Every time you validate the user, write your validation code inline with the current process. That just makes more cracking for the cracker.
D When using hard-coded keys or passwords, make them look like program code or function calls (i.e., "73AF" or "GetWindowText"). This actually works very well and

confuses some decompilers.
E Finally, never reveal your best protection secrets

1. No nagscreens. Making enemies among your customers is very stupid, indeed.
2. Create important menus and dialog boxes dynamically, whether in something like turbo-vision, xforms or m$windoze. Use your own format of rcdata like borland did in

delphi if you don't want to write too much code.

3. If your program doesn't save data in "crapware" edition, don't include a "grayed" menu item. No saving means no saving. That's it.
4. The only way to tell user that he is unregistered should be in the "about" dialog. The latter should also be created dynamically.
5. Avoid using any kind of string resources like "this crap is unregistered, blah blah"
6. Link statically unless your program is a complex one (at least of m$word 2.0 complexity)
7. Don't loose time on writing anything that will kill disassemblers or debuggers. Take my word on it - doing it is worthless, people who made them or HCU'ers :-) will soon

find the way around, so shift your interest to more important stuff.
8. Use your language of choice fully. Even if it's visual basic, learn it thoroughly first. Using some advanced constructs it is possible to make debugging a nightmare.
9. Use run time library fully when writing the beta versions, in final release rewrite some functions at least to make crackers life harder.
9a) Example: many ocx'es for vbasic take PLENTY of space, and they are really easy to hook to. Use only ones which you need, usually much of ocx crap can be rewritten to

use plain windows api.
10. Take SOME time to THINK about protecting your software. Is it worth the protection? Wouldn't it be better to IMPROVE YOUR SOFTWARE, rather than improving

protections?
11. Try to embed at least part of the protection inside the data manipulation. Data structures can take ages to understand basing only on disassembly listing, they also are

more error-prone for cracker. Crackers usually take notes on loose sheets of paper, and not everyone reads his own handwriting 100% accurately.
12. A protection that mangles data is a good one usually.
12a) Example: Got a charting program. The crapware shouldn't print. Disabling printing and later on enabling it basing on some registration# is the most often committed

suicide. Let your thingo print. When creating data structures for printing, mangle them in some way. Unmangle them just before printing, using reg# or something other for

that purpose. Even more, make this mangling subtle. Assume that you've got a pie chart to print. Don't alter anything, but add some not too big random numbers to values of

data series - this is mangling then. The chart will look "not that bad", but will be otherwise unuseable (if the changes are random and on the order of 20%, for example).

Finding such protection, if its connection with reg# is not self-evident can take much time. One has to delve inside your data structures and find that dreaded mangling and

unmangling code. Not too easy, especially if you program high level, not in asm.
13. When mangling data, use some tricks so that it is not self-evident where that reg# comes from into unmangling code.

13a) Example: I did it for a dos platform once. The reg# (in a "go/no-go" protection) was specified as a command line argument. The command line parser was clean and it

was the only place where command line was ever referenced. In some other place the "irrevelant" code was using a "bad" version of Borland Pascal move routine from rtl

(move copies a range of data from here to there). The routine in rtl was altered to copy a few bytes more and to wrap around the segment boundary. The "irrevelant" code

called copy with parameters set so that it would copy some useless crap together with command line somewhere else, and that "somewhere" was left unused for some time.

It was evaluated in the wholly other part of program. Although setting bpx on memory range would find that, it was not THAT evident that one should place the "enable#" on

command line (it was nowhere mentioned since I needed to protect from my coworkers only).
14. Be economically inclined. Calculate whether you need to write your "to be protected" program at all. Isn't there (albeit big) a commercial, widely available package that

does the same thing? If you need that tiny program for yourself only, in 99% of cases it's cheaper to use existing package that supports scritping (to tailor it to your needs). If

you want to sell it, check your market. The problem of protecting software vanishes if no one will use your software. Don't overestimate your work's "importance to the

world". Search the net. There's nothing like you want to do? Search more, then. Start work only when you're sure there's need.
15. Always explore your language of choice on the level proposed (or, more often these times, pushed) by the language developers. If you're real programmer that doesn't

want his "hello world" app to be 2megs+ big, shift down to the api level asap - ie. after exploring all the nice "visualties" the commercial types might be pushing you to use.
15a) Example: This pertains mostly to things like borland c++, delphi and m$ vc++ - write your app using the existing user interface crap (foundation classes etc). When the

innards are debugged (hah, they never are, but if you at least don't find any bugs in the first 20 minutes of post-build testing), rewrite your user interface to use windows api

directly. Do it cleverly, think first, think a lot, and you'll usually get an app that is at most 1/4 of the original size, and that works much faster. Do you believe that corel draw 3

was written using nearly pure api? Such a HUGE app? - you say it's impossible. No, it quite is, ever more in the 32bit world when you don't need to spend so much time

moving data around the 16bit segment boundary (corel 3.0 was for windows 3.x - it was 16 bit).
16. What does p.15 have to do with protections, you think. Oh well, it does have much. Since your code doesn't use the "prepackaged" stuff, it's more personal, more

custom. Since each programmer's style of programming is different, it take more time for a potential cracker to find out what's going inside your app. Alas, all this is useless

if cracker doesn't have to know the innards to deprotect. So always protect in a way that is tied in MANY PLACES to the innards (data structures etc).
17. A monolithic app that doesn't reference to much more than kernel or device drivers is usually harder to crack. There are less dependencies to get from watching the .exe

and .dll's with quickview for example. Try to export only the callbacks needed by windoze for example. Check what goes to the outside world. Try to include no or little details

of registration procedure. Your code should give as little information in "clear text" to the cracker as possible.
18. For god's sake, don't give away the working code! Try to provide users with:
- The crapware version, that DOESN'T have the "missing features" compiled in. This thing should also have no nag screens nor "enter reg# here" suff - it's obvious, since it's

the CRAPWARE ONLY.
- The full version, that HAS the full functionality compiled in but that also requires user to enter the reg# once somewhere. This version should be only given away to those

that have paid. At least there is no way to get the thing to crack from your site, since you only expose the crapware for lamers. Alas, your "full" version, maybe even cracked,

will be to be found shortly on warez sites, but not everyone knows where to search, so you at least can be proud that your app has been stolen by "the enlightened" :-)
19. As has been said many times: don't overuse the "registered", "unregistered", "registration code", etc. words - both in clear text and in encrypted form. If possible, invent

some clever algorith to generate these on the fly basing on some variables, definitely don't use the "bool Is_Registered" flag anywhere!
19a) Notes: It would be quite nice if the program will be having some sample (bad?) reg# stored inside in some place(s). The user, after entering the registration#, should

have no IMMEDIATE indication of any kind that the software is registered now. THis indication shouldn't also be deferred using timer, like in unix login. It's childish easy to

find the "sleep x" code. Each function, like save, print, etc. that depends on registration, should check the reg# by itself, and it should do it in such way that user nor cracker

doesn't see that something is going wrong or good. As said before: no disabled stuff. If saving is disabled, either don't provide it at all, or make the unregistered save work

just like real one, just that data isn't written to disk or is written badly. Consider such scheme: user isn't presented with any "this funct disabled" box, the save works ok (save

dialog opens, save completed "%" bar scrolls as usually), but you may for example lseek back to the beginning of file after writing each record if the user isn't reg#. This

approach is hard to crack since such attempt requires to delve DEEP into the program's internals and to find what is going bad and where. People also think sometimes that

their version is simply damaged and throw their program away, whereas that dreaded save might work ok when registered.
19b) Notes: CLEARLY INDICATE facts mentioned in 19a), like that: <<"Reminder: This program, when not registered, won't save nor print any data. These two functions will

operate ONLY when the CORRECT registration number is entered in the registration dialog box. Warning: we restate that save and print options WON'T WORK until the

CORRECT registration number is specified. All other features are unaffected in the unregistered version. Note: You cannot verify that correct registration number has been entered. Program doesn't display any warinings that you've entered invalid number. You know that it is ok

when save and printing works, else please retype the registration number and try again.">> This will indicate:
a) to the user that he definitely WILL enable the program when the CORRECT number has been entered,
b) to the hacker that it is going to be a tough job since the registration # checker sits somewhere else.

19c) Implementation notes: Move your reg# to the checker routine in an unsuspicious way. If you can, don't use the "existing" textbox and gettexta, implement the reg#

entering routine by hand, it would be quite wise even to refrain from using the existing text rendering procedures and render the number on-screen as it is entered using

your own renderer. You can edit your own font resource or use some preexisting font, then just encode it in some way so that it will be unreadable by resource workshops,

even better include it in the source as static constants. Although it is still easy to find where the "drawbitmap" equivalent was used to paint the reg#, it is definitely harder to

understand the whole underlying routine, especially noticing where does that reg# go. Using a multi-layered approach here is the most feasible one:
- dynamically create accelerator keys in your 'register me' dialog box, these accells should be for keys used in reg# entry (0-9,a-z for example)
- each accell should call different routine, if feasible (makes breakpointing tougher)
- each routine should store the flag that given char was entered somewhere else, it would be nice if each keypress would modify some global variable, in some way that is

decodeable for you, but not to the cracker (at first glance)
- then there should be some kind of 'monitoring' routine that acts accordingly, paining the characters on the dialog box and taking actions upon backspace and enter, for

example

- yet another routine should collect all entered characters and create a reg# and store it in some untrivial place. using a .vxd here to manipulate the virtual memory to make

some 'backup' copies in a not-easily-debuggeable way is a nice idea. a .vxd can work like a tiny embedded debugger, bpxing on the place where that reg# goes. it can then

copy it to some quite other place, all that happening in the background and not to be noticed easily. the cracker will of course (at the first try, at least) try to check where this

location is accessed. since it is accessed nowhere, he will scream: how the hell does this app know that the reg# has been entered, if it even does not access it? oh well, just

a tiny .vxd or even a background thread has copied it somewhere else. they'll get at it later, but at first it can stir crackers minds, though.
20. Notes on registration numbers:
- balance between security, feasiblity, programmability and end-user headaches
- too long, non-alphanumeric reg#'s tend to be continuously entered badly. at least provide a "non-persistent" reg# entry field so that user will rewrite the reg# each time,

possibly correctly at last. many people will just "glance-compare" the entered reg# and the one (possibly) emailed to them, arriving at the final thought that they did enter it

correctly, whereas the font is too small or they are too tired to notice that this '1' and 'l' have been interchanged (in a reg# like 'l83jjd_0)pH1lTe')
- refrain from any user feedback. the reg# entry box should accept strings of any length, without any validation. don't give crackers the knowledge about reg#. sometimes

knowing that it's 10 chars long or that is contains only uppercase alphabet helps, so don't help them
- the reg# verification scheme (I'm pretty sorry about it, but it just is like that) needs to take into account the number of prospective users, and thus you oughta do some

"market analysis"
- if your reg# is 10 numbers long, there are 10^10 possible reg#'s. but since your app might find let's say only 10^4 (10'000) users, you should invent an algorithm that

assigns each one of 10^4 users one of 10^10 reg#'s, and does it somewhat uniformly. This prevents people and programs (some .vxd based "macro" players, like the one

presented some time ago in DDJ, for example) to be used for brute force approach. If there are only 10^4 users and you allow 10^9 "valid" reg#s out of 10^10, on average

each 10th reg# tried brute-force will be valid, whereas on the case of 10^4 prospective users, that many valid reg#'s and space of 10^10 reg#s, on average only each

10^6th reg# tried brute force will be valid. Ever calculated how much time it would take to brute-force search 10^6 numbers, even using a fast machine and extremely fast

macro player (keystroke generator simulating reg# entry and checking for results)?

- the assignment operator that assigns user# to reg# shouldn't be trivial, and it's implementation should be done in asm by someone experienced both in maths (topology

known by +ORC helps here, also :-) and asm. check your operator. create graphs of how it works. understand your own work, especially its drawbacks and vulnerabilities
- be inventive. don't use anything that seems simple, quick and effective. unless you've come with something like Einstein's relativity theory, your approach is yes, simple, yes,

quick, but no, not effective, and yes, easy to crack. I'm sorry but we aren't geniuses and developing a good protection scheme takes time, it took time myself, and still takes

although i'm creating protections for fun since 5 years or so. It will definitely take some time you, dear reader, and don't believe your self-confidence. Protections written by

self-confident and unexperienced prgrmrs end up in the "most stupid protection" page of reverser. a nice and exposed place, and with what a neighboorhood, but a

protection ending there is still the "most stupid". not a nice definition of your work, huh? :-]'
21. Play same game twice. It helps. If you invent some nice and hard to crack memory move-around scheme to protect reg# inside your data space, do the same to parts of

your data, even better using the same routine just with other parameters. Crackers are lived up to the fact that protection algorithms are always used for protections only. If

integrity of your data will rely, at least partly, on proper working of your protection code, crackers get a tough work to do. it's called functional verification of the "okayness"

of protection code. you don't checksum nor crc it, but simply call it "from the other place" and let it process the protection-unrelated data. if it will process this data ok, then

cracker mustn't have altered it. this places end on easy cracks like "change that jump here and that cmp there to nops"..
22. Strainer to you: does lotus 1-2-3 for dos "diskette" protection (it let you install it up to 3 times without "zapping" it from hd first) work still in win95? if yes, why? if not,

why? analyze the code, I've learnt MUCH from it. it's only dos code, it's probably 100 times cracked already, but there are some niceties to learn from it. not a very good

protection scheme, has many holes in it, but remember what +ORC said: a cracker well educated in historical code is nearly always a perfect one :-))) (ok, ok, i know,

there's nothing perfect aside from moskovskaya in this bad world, but, can't we joke at times ? 8-0=)

Use a serial which is several KB long of arithmetical transforms, to drive anyone trying to crack it insane. This makes a keygenerator almost impossible - Also, brute force

attacks are blocked very efficiently.

Caution with the Runtime libary! Use it fully when writing the beta versions, in the final release rewrite some functions at least to make crackers life harder.
Mangle data. Protection that mangles data is usually a good one. Example: Imagine a charting program .. e.g., just disabling printing and later on enabling it basing on some

registration# is the most often committed suicide. Let your thingo print. When creating data structures for printing, mangle them in some way. Unmangle them just before

printing, using reg# or something other for that purpose. Even more, make this mangling subtle. Assume that you've got a pie chart to print. Don't alter anything, but add

some not too big random numbers to values of data series - this is mangling then. The chart will look "not that bad", but will be otherwise unuseable (if the changes are

random and on the order of 20%, for example). Finding such protection, if its connection with reg# is not self-evident can take much time. One has to delve inside your data

structures and find that dreaded mangling and unmangling code.
Traps. Do a CRC check on your EXE. If it is modified then don't show the typical error message, but wait a day and then notify the user using some cryptic error code. When

(and if) they contact you with the error code, you know that it is due to the crack. Be aware: such traps could also be activated due to virus infection or incorrect downloads.

Don't blame a potential customer for software piracy.
The rcr/rcl trick If a rcr/rcl is performed on a value, it becomes much more of a pain to crack - you can't reverse it with by negating it's effects without knowing what the

value of the carry flag was before the original operation. If the carry flag is created as a result of some other pain in the neck operation, you are probably onto a winner.
Stick conditional jumps in. Everywhere. Conditional jumps are not fun to reverse engineer. No loops, but jumps which conditionally bypass/include portions of your wonderful

key manipulation code. There is no easy inverse operation to be performed here.
Use portions of the code as magic number tables. (preferably critical sections). You have no idea how annoying this can be, if you're like most crackers and like to change

things around using softice (a popular cracking tool).
Play with the cracker's mind. This one is fun :-) Stick series of nops in, as though you were doing self-modifying code (oh my god! what the heck! nops? Aha! Self-modifying

code! Idiot spends next three years trying to find the code that should be there.). Pepper the code with junk instructions. Cut the code up into little pieces and put them all

over the executable, with (preferably conditional) jumps between them. - Anything which you would find a pain in the neck.

Detect SoftIce. Early. Now crash the computer. You can crash a pentium or a pentium with MMX even without a vxd by the opcode: F0 0F C7 C8 (illegal form of cmpxchg8b

instruction with lock prefix). Beyond that, we have to resort to the tried and true methods. Using a vxd, take the CPU out of protected mode. Windows doesn't like that.

Wonder why?

Don't rely on "EXE-packers". For almost any tool which compresses EXE files (Shrinker, WWPack32, NeoLite - to list the most popular ones) there's an uncompressor around, so compressors capable for software-protection should at least support configurable encryption. Unpackers for the above (and other) tools are not very wide-spreaded, however, don't rely on them as your program's one and only "protection"!

Protect your programs from piracy

INTRODUCTION

At one Web site I've seen a curious discourse, whether your programs should be furnished with anti-pirate devices. It went like this: No one has a right to enter your house

without your consent. The inviolability of your house is protected by law. Nevertheless, you prefer to have a lock in your front door. Further proceeding with this comparison,

we'd have to admit that after putting a lock on the door many software authors leave their keys at the front door or they lock a wrong door at all.
Here are some initial remarks.
This article is intended for those software developers who do not professionally engage in issues of protection from any unauthorized usage. It's aimed at pointing out the

most typical mistakes and helping to create more impregnable software.
Discussed here are the issues of creation of shareware and trial versions for Windows'95 (and Windows NT, if stated so). Leaving specific examples aside we can say that

everything that is mentioned below is true for other platforms, too.
This article does not claim to cover the problem in its entirety.
You can contact the author at vitas@webdon.com. Even if you don't have any specific suggestions or notices just send in your opinions about this article. This article may be

continued in future, so please give us your opinion about this, too.

There exists an opinion that hackers are unique specialists and they can be fought only by specialists like them, yet an ordinary software developer is unable to do it and he

shouldn't even try to. Therefore you shouldn't try to protect a program from a break-in, because it wouldn't save from a hacker and you should only create a simple fool-

proof protection. Meanwhile, as unexpected as it may sound, a typical hacker is not in any way an able specialist; he cracks programs because he is unable to do anything

else (a proverb «It's easier to break than to build» is true to life!). If you weren't seriously into illegal copy protection (don't waste your time if you were – in this case, this

article is not intended for you) then you are going to discover a number a number of simple, logical and very efficient ways of increasing your programs' reliability upon

reading this article (at least, I hope so). Well, of course, there are skilled professionals among hackers, and they could do a lot. The methods which are described here would

not stop them. Nevertheless, if reading this article helps you to improve your program in a few hours well enough to repel a lot of pitiful hackers – it is not so bad, is it?

SHAREWARE programs.

There are two big problems concerning unauthorized copy protection of shareware programs. The first one is how to prevent a hacker from turning a shareware version into

a complete registered one. The second one is a copy protection of a registered program, that is how to arrange that only the legal owner could use a registered copy. This

second problem is going to be touched upon in the end of this article as it requires a greater separate discussion.

Ways to create a shareware product.

When offering a shareware version of his program an author pursues two goals. On one hand, he wants to get a potential user interested therefore a program should

demonstrate all of its options and be as convenient as possible. On the other hand he wants to make a user pay for the program. The folowing methods are the most popular

ones for creating shareware programs.

A demo program.

A potential user is receiving an «inferior» program with some options completely missing, Having tried it, he's buying a normal version in a usual way. The determining fact

here is that there is no way even in theory to turn a demo version into a complete program. I'm mentioning this choice here, yet I'm not going to discuss it. Sometimes demo

versions are made via introduction of some limitations in a complete version whereas a theoretical possibility to do away with these limitations remains. Let's have a look at

two examples. A phone directory demo version contains phone numbers from just one phone station. No hacker can ever turn this demo program to a full-scale phone

directory. Another example is a demo version of a graphics editor, which won't allow you to open images greater than any predefined size. It is very probable that a hacker

may quickly remove this limitation. The difference between these two examples is that a phone directory has a part which is totally missing from it. On the contrary, in the

second example there is a removable «extra part».
Incomplete functionality
A shareware version does not include some useful functions which only appear after the registration. It's very important to determine what functions are going to be

inaccessible during the trial period. At the present time the software market is extremely satiated. For most of the programs there exists an alternative by a competitor. So, if

you deprive a user of many useful functions he may not want to try your program any further and buy it. It is well known that, when meeting a new person, one's attitude to

him (or her) is formed during the very first seconds of the acquaintance. The same thing happens to items and programs – it is very important that your program impresses

a user as much as possible. Therefore, the missing functions must be useful but not essential for the first acquaintance with the program. Moreover, these functions must be

evident and intuitive, that is, a user should clearly see what exactly is going to be at his disposal after the registration and how convenient his work is going to be. During the

trial period a user should get used to your program, get familiar with it and want to use its at its full power and at any time. Therefore a choice of options to be removed for

the trial period is very important.
Date of usage and launch count limitations.
It's a very popular method of protection. It does not have a drawback of a previous method – as soon as at a time of trial your program is going to be presented at its best

for the user. This method is accepted in beta versions of many programs to limit the time of their usage (testing). Even the well-known Microsoft corp. uses this way of

testing.
Annoying behavior.
During the trial period a program does something to annoy a user but not to prevent the usage of all options of the program. Usually it is a reminder about a necessity of the

registration. A good example is a popular WinZip program which recommends to register itself after every launch. This variant differs from the first one by the necessity to

«add» something in case #1 and the necessity to «remove» something in case #3. It is evident that the third method is outperformed from the point of view of reliability.
Registration info.
This option is intended for honest people – a program would just inform them that it is not registered yet or who it is registered for (on-screen, in a demo or in an About

box). This way of protection is present almost in any program, but usually it is not the only one.
Upgrade.
A new version of a program should allow using itself only if a previous version of this program is already installed.
Program registration.
How a program is being registered? Of course there exists an option that, upon paying for the program, a user will receive a new (or complete) version in a glossy box

bundled with complete documentation, possibly delivered by a parcel service. In this case we'll have to speak about a demo program which is followed by a purchase of a

complete version, yet we have decided not to discuss this case. Another drawback of this method is that a user does not receive a result immediately following a money

transaction (for example, paying on-line with a credit card), having to wait for it several days (at best). Moreover, an Internet program distribution without traditional boxes,

floppies and manuals allows to cut expenses down and to increase efficiency, which is so important for various utilities and other inexpensive programs. Upon a registration a

user is naturally supposed to receive something that can turn a program into a registered one from a program manufacturer. Usually this something is a registration number,

but let us understand it widely – it does not have to be a number, it is a piece of information given to a user by a program distributor. Given a chance, you should make this

number readable and not too long. Consider, for example, entering such a registration code:
04846718351B3E6B3D717E240A200A29556B7C18347A1162526F5C This real-life

example (well, maybe almost real-life :-)) is a registration number from a program named GoldEd. You should make it more readable, that is, use only digits and separate

groups of digits with optional separators, like this:
214-527-822-655-121. If a program is registered on-line then a problem sorts itself out because a registration process can be automated.
Ways of cracking and ways of protection.
Most of shareware programs can be cracked so easily mostly because their authors cannot imagine what methods are use by hackers for cracking. Below we are going to

discuss the simplest and the most frequently used ways of program cracking and ways of protection.
Cracking programs with date limitation or launch count limitation.

In case of limitations that are imposed on period of program usage or times a program had been launched some specific cracking methods become available not only for

hacker, but for smart users, too. One of the ways to cheat a protection goes like this: before installing the program it is necessary to set a date several years ahead, for

example, at 12.31.2017. Let's say your program only works during two weeks after having been installed. Then it's protection system would decide the program is supposed

to work till 01.14.2018. Having installed the program, a user reverts to a correct date and uses your program till the end of the world. How could one counter such a method?

The first palliative is to check a current date and, if it is too big (say, more than six months since program release), suggest to use a newer version. A more radical way is to

check a current date every time a program is being launched. If, suddenly, it turns out that a current date is lesser than the installation date, then… At this point you may be

tempted to display a message which accuses the user of cheating and maybe even to uninstall a program. DON'T DO IT! The user's system might just have a time failure.

Alas, if a time failure did happen, your program will stop working short of its time. This is a rule of thumb – increasing protection reliability leads to increasing your program's

discrimination. A program can be made to behave a bit smarter – if system time is lesser than the program release time then the system clock has really failed, because

when a failure occurs a date is often set to 1.1.1980. In this case you can suggest a user to correct the system clock.

The second way of cracking lies in setting a wrong date at the runtime and restoring a correct date afterwards. Of course, this way is a bit awkward, but, unfortunately,

there are programs which help to automate this process. It is extremely hard to counter this method. You can try the following ways to do it.

It is a radical way which can be afforded only by developers of Internet-based programs. Since a user is supposed to work online, your program can get a current date and

time from a server that provides with exact time and date; there is quite a lot of them now. At the same time your program can do a good service by synchronizing the

system's time and date if they are set incorrectly.

Unfortunately, this way is acceptable for very few programs. It can be slightly modified; if a computer works in a local network, a date can be obtained from a server or from

other computers.

If this way is unacceptable, either, we have to admit that without leaving the bounds of a computer, there is no way your program can find out what date it is. Yet, a fact of

altering a date still can be detected. Many programs, for example, record their relevant information into the Registry. For this reason, the registry modification date usually

matches the current date. When Windows is started, the system creates backup copies for the files c:\windows\system.da0 and c:\windows\user.da0 (these files are marked

as the hidden ones).

You can check the date of their last modification and, if it turns out to be greater than the current one, then a user must have set the clock to an earlier date. It is important

to check the backup Registry files because the current registry files may already have been modified and therefore their modification date may match the current wrong date.

Evidently you can not only check the Registry files like this, you can also verify your own data files. Here's how a real current date may be obtained: as you may or may not

know, Windows' 95 not only stores the creation date/time for each file, it also stores the time and date of its last access. As a true current date you can consider the last

access time of a file which is always launched or opened once when Windows start up. Autoexec.bat or windows\system\vmm32.vxd may serve as examples of such files.

The third and the last way of protection that we are going to consider consists of installation and uninstallation of a program. Modern programs usually have convenient setup

utilities which install and deinstall programs. As a result, it may be not burdensome at all to perform a simple installation procedure once a month. There are different

methods to fight this method. If, for example, your program requires a great deal of setting its options, you can arrange it in such way that uninstalling a program would

cause all options to be lost and a program would have to be set up again after the repetitive installations. A more radical solution would be to leave a mark in a computer (in

a Registry, for example) which would state that a program had already been used on this computer and to refuse to install a program if such a mark exists. Indeed, your

program would have to be a true perfection so that a user would want to reinstall Windows just to be able to use it. Should I mention that this mark must not be removed by

an uninstaller, it should not be located close to other settings of your program and the last, but not the least: use your imagination! If your program is named CoolApp.exe,

do not name the register key for its mark as CoolAppWasHere! An alternative of using the Registry may be a small DLL in the Windows directory which an uninstaller does not

remove.

If there is a limitation of a launch count, then a counter must be decremented after each launch. (It may be incremented – this is totally irrelevant for us. Let's suppose it is

decremented, just to be certain.) It is very probable that this counter is located in the Registry or in an ini.-file. At times, it's just amazing to see how clever people who are

able to create brilliant programs show miraculous narrow-mindedness and name this counter, for example, iTimeSpan, like it is named in the FineReader program. Name

this key variable, for instance, RAS32stub and make its value equal to the remaining launches count with a checksum, encrypted in any code, even a trite XOR. Although this

may not help you a great deal – if a hacker finds the counter, he can just restore its original value every time. A radical remedy which is acceptable for Internet-based

applications is keeping your counter on a server instead of a local computer.

It is possible to store into a parameter a checksum of the remaining parameters, including the counter itself. In this case all the settings of the program can only be restored

together.

You can try and hide the counter outside the Registry, because this location is too evident. You can come up with an impressive name for an .INI file (which, by the way, may

have an extension other than «INI.»), stuff this file with a load of settings with intimidatingly complex names (the counter is to be one of them), place it in the Windows

directory, and, more importantly, every time you decrement the counter, you should forcefully set the same date for this file, 08.24.95, for instance. I hope you have been

already influenced by the anti-hacker spirit and understand, why I'm suggesting you to use this date.

Comparing the time-of-usage and launch-count limitations we'll have to admit that the second option is less reliable because of the necessity to keep a constantly changing

counter which can be found with relative ease.

Methods of hacking without code modification.
Suppose a registration process involves sending a registration number as a response to a user name. All too often, just after a program is released a key generating

program appears. Such a program is able to generate a registration number for any given user name. Meanwhile, it is not hard to arrange everything in such way that

creation of key-generating programs would be completely impossible. To accomplish this, it would be sufficient for you to use a simple asymmetrical encryption algorithm.

You can find a lot of information about asymmetrical encryption algorithms, often dubbed as open-key systems, for example, at www.rsa.com. To put it in a nutshell,

asymmetrical encryption algorithm is a way of coding information which requires one key for the encryption and another one for the decryption, yet it is impossible to obtain

both of the keys, knowing just one of them. As a rule one of the keys is made as an «open» one (which one of them it is depends on the purpose of their usage), that is, a

commonly known key, the other one is kept as a secret. Now consider this: a registration number is a user name which is encrypted with an asymmetrical encryption

algorithm, and the encryption password (a key, that is) is known only by you. A program would perform the verification by decrypting the registration number (the decryption

key is built into the program) and comparing the result with the user name. If such an algorithm is used correctly, it is totally impossible to create a key generating program.

It has to be noticed, however, that the aforementioned way of protection will not prevent from using someone else's key and from cracking by modification of program code.

This method is essentially an electronic signature with a registered under name by the program's author.

Most of modern commercial (non-shareware) programs require entering a serial number during the registration. This would pose no problem for a legal user – the number

can be found in the program package, and could pose a problem for an illegal one (if we aren't talking about a pirated CD, that is, because thoughtful pirates often place a

file with a required serial number on the same disk). A serial number has to satisfy several criterias therefore a random number will not work. Yet all too often an

implementation of checking the number validity cannot stand any criticism. For example, most of Microsoft products can be satisfied with a number which consists only of 1's.

Here's a method of verification which can be advised (actually, there can be quite a lot of options): the first part of the serial number is encrypted (even in the most primitive

way), the result checksum is calculated and then a remainder of division of checksum by some number X is calculated. The resulting remainder should match the second part

of the checksum. No matter what kind of verification you use, at least it should not miss banal numbers like all zeros or all 1's. It would be a good idea to provide a link

between a serial number and a program name and/or version to make it impossible to use serial numbers from one of your programs with another one. One quite successful

brand of this method of protection is a method which requires entering word # N from page # M from printed manual which should be provided with the program in this

case.

Here's a bit about upgrades. A newer version of a program which allows to be used only if there is a previous version of this program already installed on the computer must

have some mechanics that scans for the previous version. In a likely way Windows-95' upgrade requires presence of Windows 3.1 during the installation. Yet, alas, all of the

verification consists of checking for presence of a file windows\system\win386.exe. It is enough to create a file with this name (you can even make it 0 bytes long) and

Windows '95 can be installed on a computer where there is no and has never been Windows 3.1 present. Hence the verification should be a bit more complex. For example,

an upgrade version of a program may purposefully omit setting several options in the Registry, provided they match those in the previous version. It is possible to come up

with a lot more of highly reliable methods.

Methods of hacking involving code modification. Very often hackers turn a shareware program into a full-scale working one, having changed just one byte in its code and

having spent just a few minutes. How do they do it? How do they manage to find a right place in a multi-megabyte program and spend less than a minute on this search (this

is genuine time it takes, not an exaggeration)? The matter of a fact is that authors of programs seem to do their best to help hackers in their work. Let us discuss one very

typical example. Suppose your program goes something like this:

```
BOOL isRegistered() {
...
return TRUE or FALSE;
/* this function may determine if this program is registered or not, basing on some probably very complex and perfect criteria*/
}
....
//a warning for a user
if(!isRegistered()) MessageBox("This program is not registered!", "Register please!");
```

....
// here's a verification of being registered when performing some useful functions if(isRegistered())
doSomethingVeryUseful();
else
MessageBox("Unregistered copy! This function is available for registered users only!");
....
//there's a lot of such tests, and they are stuffed all over the program in various modules
The most funny thing about this is that many authors actually find this kind of protection very reliable. Well indeed, the isRegistered() function is very complex and the

verifications are scattered all over the program in abundance.
What is a hacker going to do to such a program? That's right – he will just find a string, for example, «Register please!» and get its address. Now it's not hard to find a piece

of code like this one:
call xxxxx ; function IsRegistered
or eax, eax
jnz yyyyy ; *
...
push <found_address>; This is the place a hacker will find.
....
call zzzzz ; MessageBox
Now he is going to alter the only byte in the code, the one marked by an asterisk and there will be no demand of registration any more. Sometimes it's enough to put a

period here. In this case the time of hacking can be measured in seconds. Yet we are examining a more complex sort of protection – this testing is performed repeatedly. Get

rid of an illusion that a hacker will try to find all of its occurrences (although it is not that difficult). He is merely going to alter a few starting bytes of a function IsRegidtered in

such way that it is always going to return a TRUE value. This function may implement the most brilliant and the most reliable algorithm – but who needs it now?

Pondering over this given example we can reach two conclusions. First, the hackers' task would get much more difficult, had he not found this place in the program so easily.

Secondly, we cannot live with the fact that there is a theoretical chance to break a program by replacing just a few bytes. The right place was found only because the right

string was found. Of course, this is a specific case. Instead of text there could be a resource id, if a message is stored as a text resource or a dialog box. There could be

some distinctive number. Suppose an unregistered modification of your program limits some of its setting by 20. In this case a hacker might be looking for the code

cmp <something>, 20

To hamper the search of the necessary place for a hacker you should diguise such critical segments. Text strings (such as messages, filenames, Register key names etc.)

should be encrypted for storage (in any, maybe even the most primitive way). The telltale constant numbers should be disguised like this:

```
#define SLY 21061969
int vSly = SLY;
...
LoadString(IDS_SHAREWAREWARNING - SLY + vSly);
...
```

Besides, the verification is not to be carried out in the very beginning of a program. It is preferable to have several testing procedures and perform them repeatedly. You can

create a separate thread which performs such necessary check-ups periodically.
Let us take a more global look at the problem. How to achieve that your program theoretically cannot be broken by altering just a few bytes? Yes, it is possible. Consider an

example:

```
struct VeryImportant {
/*
```

Collect all the constants you use in functions which are not available for unregistered users into this structure. Accordingly, in the functions themselves use the constants upon

retrieving them from this structure. For example, instead of a = b + 2;

write a = b + myconst.two;

Let there be just a few of your constants. They are going to be important and non-ordinary, if possible. A pointer to a function can be used as a constant and, if necessary,

this constant can be used to call this function by this very pointer.

```
*/
} myconst = { xxxx };
/*
```

Here is the most cunning part. xxxx are the constants we need, encrypted in some algorithm. Do not re-create anything that is well-known, use a professional algorithm, for

example, rc2 or rc4 – they are not complicated at all. The necessary values may be read from a file, a resource or extracted out of a Registry (where they are put by an

installer). The last option is the most attractive one.

```
*/
....
```

```
//This is an unencrypted checksum of our constants.
const MyConstCRC = xxx;
....
//here is our most curious function. It is very simple now!

BOOL IsRegistered()
{
return CalculateCrc(&myconst) == MyConstCrc;
}
....
//this is where the registration number or zeros, if the program is unregistered, are stored.
BYTE regNum [REGNUM_LEN];
.....

// this function should be called once for initialization
void initProtection()
{
// now we decipher our constants with the registration number (or, rather, with its part)
decrypt(&myConst, regNum);
}
// The following part may be just like the one in the previous example.
```

Even now a hacker may find the function IsRegistered(). Now modifying it will only make your program stop working, because the constants which are located in the structure

and are necessary for the work cannot be deciphered without the registration number. In other words, you should follow a simple rule – a value being verified should be used

instead of being compared to a constant. So, we do not check for a presence of a registration number, we use it. Nowhere in a program a hacker may find what value this

number should have. Let's view one more example of this concept's usage. Suppose an unregistered modification of your program allows not more than 40 records in some

database. To simplify the matter we suggest that each user is described by a structure named CUser, and the database is just an array of the CUser structures. Usually there

is something like

```
array = new array[userNum];
...
if(userNum > 40) ....;
```

present in the program, and a hacker may simply find an operation

```
cmp <???-??>, 40
```

and alter it in the way he wants.

Having learned the aforementioned, you can make his life a bit more complex in this way, for example:

```
if(userNum*2+1 > 81) ....;
```

But you can go further:

```
CUser *array;
// in a registered version initialization is performed by another function
void initUserArray()
{
int x = -83, y = 3;
array = new array[(x+y)/-2];
}
...
if(userNum*2+1 > 81) ....; // *
```

In this case, even if a hacker finds the verification (marked by an asterisk), he is going to get himself some very unpleasant and hard-to-find range check errors when the

quantity of records exceeds forty.

It is harder to implement a protection if an unregistered program is supposed to display reminders. This variant is different in that a hacker is supposed to remove an «extra»

to get a full-scale working version, not to add a missing function. To break is easier than to build, and this applies to this case, too. You can protect your program from

cracking to some extent, if your program does something minute but necessary for your work when displaying a reminder. For instance, a reminder is implemented as a

dialog box. When the WM_INIT message handler (dialog initializer) is executed, a timer can be set to one second. And when the WM_TIME timer handler is executed, you

can initialize some necessary variables or perform another action which is necessary for the further work. If a hacker finds and suppresses the window output,

simultaneously he renders the program inoperable.

Another evident way to protect your program from cracking is to check its integrity. For example, your program can count its checksum upon loading (the checksum of a file,

not in memory). Although this cannot boost the speed of operation in any way, it can also help you to locate a virus infection or a violation of a program's integrity which can

occur during the program's transfer via communication channels. It has to be kept in mind that, generally speaking, the integrity verification cannot guarantee the integrity

itself, because a hacker may remove the verification subroutine itself. This is where the «chase» begins, whoever succeeds first. The integrity verification can be logically tied

to a digital signature of a program (a.k.a. its certificate). There has been written a lot about it, so we won't talk about it here.

Copy protection.

Let's address another problem now – how to prevent a distribution of the registered program? Generally, its a problem of a «traditional» copy protection. There is a lot

written about it. Listed below are the most popular ways of protection.

Key disk.

This way of protection is practically not used any more. Firstly, it is very inconvenient for a user, secondly, it is unreliable because of unreliability of disk as an information

medium. By the way, a common belief that this method is unreliable because any disk can be copied with a special program, is not true.

Electronic key.

Aside from some apparent benefits, this method also has its disadvantages. Usage of two programs protected this way on one computer may turn out to be problematic.

Because of a sensible cost, this method may be inappropriate for inexpensive programs. Besides, this method is only potentially reliable. In practice, however, we may

encounter some very clumsy implementations (here's a specific example: a verification function is located in a separate DLL and returns a true/false value «key present/ key

not present».

Hardware binding.

Binding a program to equipment seriously limits a user, because a computer upgrade becomes impossible.

Software binding.

This binding may not be evident because program which has a complicated installer that carries out a large preparation work (entering settings into the Registry, installing

common DLLs into the system directory, registering ActiveX controls) essentially binds a program to a specific copy of software installed on your computer.

Conclusion.

If your program is distributed on CDs and its volume is not too big (that is, a program does not take up a whole CD), than you may consider filling the free space with the

information which is needed for the installation. If you do it skillfully enough, than pirates would have a question arisen – is it worthwhile to issue a CD which is completely

dedicated to your small program?

Well, pirates strive to place as much of different programs on a CD as possible, in order to increase the quantity of potential buyers. Your program may be small, yet have a

bonus pack of useful, yet non-essential programs included with it. Pirates can easily «crop» your distribution package, therefore your installer should check for the presence

of this extra information. You can also check if some files or directories previously absent from an original package have appeared on a CD.

And, just a little notice in conclusion. You can dramatically increase your program's fortitude, yet it is going to take greater carefulness and attention than common

programming takes. Here's a small example: when proposing a method of protection by means of setting a counter of program launches, this article suggested to put this

counter into a .ini file with a name that sounds intimidatingly scientific, and the most important thing was to forcibly set the same old date of modification for this file after

each decrement of the counter. I hope that, upon reading this article, you will not store the name of this file as an open text. To manipulate this .ini-file you will probably use

a WIN API function WritePrivateProfileString. Do you remember that Windows caches .ini-files to increase the operation speed? Here's where such a situation may occur: you

use the WritePrivateProfileString function, yet saving an .ini-file is delayed. You then set a date you need for the file. And then the cache is being dumped onto disk with a

correct, yet undesirable date of modification specified. How to counter this problem is mentioned in the description of the mentioned function – I used this example to show,

just how accurate a programming should be.