



**A Developer's Guide
to Software Protection:
*Applying Modern Coding Techniques
to Securing Software Assets***



Introduction

Purchasing software protection products such as hardware tokens can provide you with a very secure lock, but you have to be sure not to leave the key under the mat. A secure implementation is vital to ensure your application has the maximum protection your anti-piracy technology can provide.

Software is unique and we cannot write a perfect implementation formula that can be effectively reproduced for every application. Skilled developers with comprehensive knowledge of your software are the people best suited to execute a secure implementation.

Fortunately, there are a few general tips we can still provide that may be helpful and applicable to your software. Not all the techniques discussed here apply to all development environments/compilers. Some work differently (eg. on intermediate code like Java or .NET), others don't have marching, etc. However, much of the generic advice here is applicable to all protected applications.

Don't Write Your Own Protection

Most software protection aims to restrict the use of software to some limited license conditions. This can be done via hardware tokens, software-only licenses, or some homegrown code. Software developers – except in companies dedicated to software security – often have little experience with software protection. They don't know how crackers work and how to work against them. Just like in cryptography, non-experts may attempt to create new methods and crackers can have an easy time breaking their protections.


Software security vendors have dedicated teams of experienced individuals. They have seen large numbers of cracks in the past and understand how crackers analyze, disassemble, debug, patch or emulate software. Security vendors may also frequently test-crack to make sure they are familiar with the latest disassemblers, debuggers and other cracking tools. This expertise enables a skilled vendor to protect software and make a cracker's life harder, so the effort required to break protection is substantially increased.

Do Not Check Only for the Presence of a License

To protect software, somewhere a licensing condition needs to be checked. If a valid license (hardware key or soft license) is present, the protected application is allowed to run; if there is no license, then the application should not run – or run in a restricted mode.

Many developers try to get around the protection by, at program start, calling a function to check the license condition. This function may be either a protection API call or the developer's homegrown checking routine. Either the function does not return if there is no license (somewhere in the subroutines it uses, the application is terminated) or it returns with a "Yes/No" return code. The latter may be Boolean TRUE/FALSE or an integer where zero usually means "OK", non-zero means an error. Crackers can find the routine and simply change it to immediately return "OK". With that patch, the program will always run and won't even bother to check for a license.

An indirect verification of a result is a better check. If C=Challenge value and R=Query(C), the response from a protection device then, instead of comparing R to the expected value, we can check some *characteristics* of R, eg. number of '1' bits, sum of the bytes in it, remainder if divided by a constant D, etc. Even better, it can use that characteristic later in the program, such as an index to some function or data array.



Crackers frequently run the program side by side on two computers, one with the proper license and one without; then see in the debugger where the code diverges.

Good protection codes must ensure that such simple patching does not work (ie. without a license the application will not work correctly. It may still work – just incorrectly, possibly producing invalid data).

Hide Calls: Get Function Addresses Directly from Loaded DLLs (Windows)

Crackers often analyze what functions an application is using and where. In Windows these are API calls that reside in system DLLs (eg. kernel32, user32, advapi32). The Win32 executable has an import section that lists the function names. A cracker can learn a great deal just by looking at the import table: such as whether there is registry use, CryptoAPI calls, etc. By disassembling the code it is also easy to see where such calls are made (eg. where file opens are called). This helps debugging and analyzing the code to find out where the software protection related routines may be in the application code.

One way to combat this is to obtain the function addresses at runtime so they will not be listed in the import table. Another simple way is to use LoadLibrary and GetProcAddress calls, but these can also reveal what you are doing. Instead, one can obtain function addresses by directly searching for them in the DLLs' export table. It is a bit more programming but it is well worth doing.

Here are the steps to get a function address directly from the DLL image:

1. DOS header <- DLL load address (from LoadLibrary)
2. PE header <- [DOS header + 0x3C]
3. Export Tbl <- [PE header + 0x78h] + load address
4. Look up function name in array, then
5. Get function address RVA from parallel array
Function address = RVA + load address

Check if External Functions Have Breakpoints

Crackers try to find out where certain functions are called and with what parameters. If they can't find all references from the import table (ie. someone is obtaining function addresses directly from the DLLs) then they can still debug the application and set breakpoints to the APIs they are interested in. (eg. CreateFile – to see when the filename is that of a license file or device driver)


An application can check if APIs have breakpoints (usually INT 3 = 0xcc code byte). If yes, it is an indication that the program is actively debugged – and may be time to just stop running.

On Intel x86 machines crackers may use hardware debug breakpoints. They cannot be detected by a simple code check because they do not rely on the INT 3 instruction. Instead, they use the processor's debug registers. However, there are still techniques to either disable them or detect that debug registers are actively used on the running application.

Hide Strings

Even if one employs the trick to get function addresses directly from DLLs, the function names still need to be checked. If the developer just uses string constants (eg. "RegOpenKey"), those can easily be seen in the code / data area and can be traced back to where they are referenced.

There are many ways to hide strings. One can encode them, then decode when the string is needed, and re-encode it after use. Another way is to build the string character by character, even in a re-ordered sequence, into a temporary area (eg. local variable) which is then destroyed or overwritten after use.



Naturally, not only function names but also most licensing related strings (eg. file names, error messages and the like) should be hidden from direct observation of the code; both in the executable file and in-memory after the program has started.

Don't Use the Same Functions Everywhere

Developers try to increase protection by not just checking the licensing at the start of the program, but later on periodically. However, they often use the same function called from multiple places of the application. Obviously, if the function is patched out than it doesn't matter how often the program tries to check the license, as it will always return OK. It is better to create multiple functions with slightly different details so a cracker cannot easily find all of the places where license checking is done.

Note that it is much easier to debug the start of the code than debug *all* the code in the application. License checking can be triggered by many actions in a program and, if the application is sizable, than debugging all the code is infeasible. If the application has separate modules (eg. in Windows several large DLLs), another option is to put some license checking into each module.

Spread Out Checking Routines in the Code Section

Most developers treat protection code just like any other part of a project. That usually results in all security related code residing in one or two source files. When the application is built, the routines will be in one continuous code area. Crackers love this because they don't have to look around; once they find some security related code they just have to look at surrounding code to find all relevant code.

From a project maintenance and software engineering point of view, the practice looks good. However, the fact is that we are not just creating "normal" code. The goal is to hide the protection code as much as possible. This may require spreading the software protection functions all over the code base. In this case, be sure to maintain an up-to-date reference which states what security related functions reside in what source code files.

Use Silent Code


Silent code is a sequence of instructions that runs entirely within the code part of the protected application; it doesn't call outside DLL or system functions, device drivers, or interrupts. Silent code is most difficult to locate if it is not directly preceded or followed by code that calls outside of the application's code. Important security code parts should be put into such silent code blocks, then these code areas should be put into "normal" application (ie. not security related) code.

Calculate Results – Then Use Them

Software protection hardware tokens may allow the calculation of some values based on secrets stored inside the keys. Encryption algorithms may be used to return a response to a query. Anywhere from a 32-bit to 16 byte value can often be sent and received.

Communications between the hardware key and protected application often use two tables: an input query values table and an expected-response table. Tokens can send a value from the first table, get the response, and compare it to the expected value (obtained from the second table). Although this is an improvement over the simpler "is the key there?" check, it still is just a compare and yes / no check ("Is the compare OK? Yes or No") If a cracker finds the compare instruction in the code it can be changed to always give an "OK" result.

Instead of this simple comparison, the code should get back the response then – much later – use the obtained value for some important internal calculations. It is not difficult to make the returned value appear very random (and what can a program do with a 'strange' value?)



Assume somewhere the application needs to use a constant value C for some important calculation. The developer can select any fixed input value X, send it to the key and obtain back the value Y. (Some query algorithms X and Y may be only 32-bit so we can use them as unsigned integers. In case of AES response, we can just use any 4-byte part of the response as the Y value.) Now calculate $C^* = Y \text{ XOR } C$ and store X and C^* in the program code ('hard-wire' the values or put them into a file). At runtime, well before we need the constant C, we send X to the key and retrieve the response value Y. A bit later we calculate $C = Y \text{ XOR } C^*$ and store it somewhere; and when we need the value now we can use that constant. Note that if the cracker just patches out (or ignores) the routine which sends X and retrieves the value, then the program will never get the correct Y value and will calculate some bad C value which will result in bad program behavior. It will be much harder to trace the problem because there is no obvious "yes/no" place.

Check Part of a Result – Then Crash if the Other Part is Invalid

As a variation of the above method, when you get back some longer result (eg. in case of AES it is 16 bytes) you can then compare part of it (eg. 8 bytes) with the traditional query/response table. The cracker may find the compare place and patch it out. However, unbeknownst to him, the code will later use the other half of the result to do some calculations and derive some important constants. Now it will be even harder to understand why the application is not working as expected because the cracker believes he solved the query/response check.

Another option for using the rest of the returned value is to do some calculations in multiple steps – which should result in zero. If the result is non-zero then jump to the calculated value (as address); this will result in the application just crashing. (Finding such a place in the code is very hard: normal application code often checks if something is zero and does things only when it is not, such as with memory allocations, function pointers, etc., so this tricky code would look legitimate.

Hide Access to Variables (via Array Indirect Access)

You may have data that is accessed from various functions. This will be the case, for instance, if you get some data in one routine (query/response or AES) then use it later somewhere else in a calculation. Crackers have to find where that data is used and look at how it is used. Normally this is an easy task as the same data is referenced everywhere by its address – and that address is visible in both the binary code as well as in the disassembly list.

Since our goal is to hide *where* we access the data, we can do it with some tricks. Let's assume we have an integer variable:

```
int g_mySecurityVar;
```

Normally, everywhere we use it we just reference to it as "g_mySecurityVar," but that would result in the code being always generated with its address. Instead, we can choose a large array size SIZE (eg. =50) and declare the variable as

```
int g_mySecurityVar[SIZE];
```

When we want to reference it we use a local pointer variable

```
int *pMySecurityVar;
```

then do the following trick: First, we set the pointer to the address of some elements of the variable

```
pMySecurityVar = &g_mySecurityVar[X];
```

(where $X < \text{SIZE}$ eg. $X=37$), then decrement the pointer X times. Note that the latter may be tricky, as good compilers often figure and optimize out such convoluted code. Usually, we either need to add some nonfunctioning assembly code or use a function that does the decrementing, eg.

```

int * DecrementIntPtr(int *p, int n)
{
    while (n--)
        p--;
    return p;
}

```

(simplified code, no NULL pointer or boundary checks shown)

Using such a function has a drawback: the cracker can see where such a function is called, and that gives away the security code places.

Manually adding this type of code is tedious; moreover, we would like to make sure that different places use different X constants. To take care of the latter in places of such constants we can use “__LINE__ % SIZE” which will automatically generate different values. An added twist is that sometimes this will be zero so in a few places we may actually refer the address of the actual “g_mySecurityVar[0]” value. This may seem ineffective, however giving the cracker *some* address references may make him think he found all the places we use the variable. Moreover, with this re-write we can easily create macros which now give us access to our security variables in a way that the generated code will not show the address of the variable.

Access Variables via Allocated Memory

Even if we re-write the code so it will not show the address of data we use, a cracker can use memory breakpoints in a debugger to find out where we use the data. One way to make it harder to do such tracking is to use allocated memory. If done correctly (see “Allocate Random Amount of Memory at Startup,” page 7), the memory address of the data will be different each time the program is started. This will prevent the cracker from easily putting the memory breakpoint on the same data each time it starts the application to debug it. One can also allocate new blocks of memory for the same data, forcing the cracker to manually change the debugger breakpoint address at each re-allocation.

If we use allocated data, we won’t use the indirect-access array trick above on the data itself; instead, we will do it on the allocation pointer. So now we will have

```
int *pg_mySecurityVar[SIZE];
```

then we allocate the memory

```
pg_mySecurityVar[0] = malloc(sizeof(int));
```

whenever we want to use our data now we do it via local pointer-pointer

```
int **ppMySecurityVar = &pg_mySecurityVar[X];
```

The rest of the method is similar...


Copy Variables to Multiple Places

One powerful debugging tool crackers often use is the Intel x86 processor’s debug registers. Crackers can set memory breakpoints and monitor when a given address is accessed (read) or the data is modified (write). This helps in two ways:

1. The cracker can monitor where and when certain variables are accessed
2. If the cracker finds some data of interest it can help back-trace where it was created and from what other data

One limitation of this tool, however, is that there are only four debug registers, so at any one time, a maximum of four addresses can be monitored. We can turn this limitation to our advantage: we just need to use many more variables so the cracker cannot track them all. One method, mainly for read-only data, is to copy the data into many locations; then the different code areas access different copies. This makes it much harder to track all access to security related data. If we combine this with accessing the data via allocated memory, the cracker cannot even try to cover all access with running the application multiple times.

For data we manipulate, it is even more difficult for a cracker to trace. Use multiple allocated memory blocks to hold the data, then copy it over to another allocated memory block, then another allocated memory block, etc. We can manipulate the various data



copies differently, so not only the addresses but also the values will diverge. If all important data is held in a large number (8-10) of different addresses than the cracker has little hope of tracking them all. If we do this with many variables, then the four debug registers will severely limit the cracker's debugging ability.

Access Global Variables Often

Another way to make tracking variable use difficult is to simply use them frequently. In this case, the cracker will become tired by the constant hits in the debugger. Note that for this type of access we really don't need to hide the address, as these access points will be everywhere in the 'normal' application code where nothing important happens to security.

Depending on the use of the data we may want to hide where it is set/modified. In that case, we may need to write to the data (not just access it). For integers, one way to do this is to declare global zero value data

```
int g_zero = 0;
```

then add this to each variable we want to access

```
g_Var1 += g_zero;
```

```
g_Var2 += g_zero;
```

If we just need to 'touch' the variable (ie. our concern is more of read access tracking via debug breakpoints) we can use a dummy variable to 'collect' the values

```
int g_ignored;
```

and the access line will be

```
g_ignored = g_Var3 + g_Var4 + ... ;
```

Similar access code can also be written for character and floating-point data, too.

Use Macros – Generously Sprinkle

Since we want to access the variables throughout the application code, it would be hard to add the same code everywhere manually. Fortunately (at least for the C family of languages) we can create macros eg.

```
#define TOUCH { g_Var1 += g_zero; g_Var2 += g_zero; \  
              g_ignored = g_Var3 + g_Var4 + ... ; }
```

then generously sprinkle "TOUCH" lines everywhere all over the source code. If you have some tight code where speed/performance is important, you can leave out such places. Generally, this will be a very small portion of the code, so the macro can still be put to hundreds of places. Macros can also be employed to ease the use of indirect access to security related data.


Allocate Random Amount of Memory at Startup

Crackers usually have to run and debug a protected application multiple times. If each time the application runs exactly the same and all addresses are the same, it is easy to debug the program and can even be partially automated. One simple way to make sure at least some data addresses are different each time is to allocate a random amount of memory at the very beginning of the application. From that point on, all other allocations will give different addresses, making debugging across sessions more difficult and more time consuming for crackers.

Create Variations of Checking Code

As discussed, if each time the application starts, and the same code is executed, crackers have an easier time debugging it. One way to counter is if we have multiple ways to check the license condition. These routines may be the exact same code or may be slightly different. Depending on some runtime-specific value, we may call one or the other routine. That way when the cracker patches out one checking function, the code will call a different, unpatched function at another time and the program will not work.

One selection method is total unpredictability; the code selects the function to check based on time-of-day or process ID or some other runtime value. The downside of this is



that the cracker will immediately see it because when he patches out one routine, the next time he will see that the code is running on a different path.

A better way is to greatly delay such difference – eg. use only the month as the selection vector. When the cracker patches the code (at least the code path for the current month) it would seem that the application now runs without a license. Even end-users downloading such cracked application will be happy that they can run the program without paying. When the next month comes, suddenly, nothing will work – all “cracked” copies will start to demand licenses again. Since crackers usually shy from giving away their identities – they just post patches to various wares Web sites – it can be difficult and frustrating for an end user to try to contact the cracker and ask for another working cracked version of the application.

Another option is to calculate the selection vector based on the computer on which the application is running. We may use disk label, system directory date/time, username, etc – or some combination of these. In that case the application always runs the same way on the cracker’s computer – but will run differently on an end-user’s computer. The cracker may not see code paths that will become active on other computers. (Even if the cracker has more than one computer if we use a larger number of code paths he may see two or three – but not all.)

A simple way to decide on the code path can be multiple ifs (or a switch statement) like the following pseudo-code:

```
if month == January
    call routine1
else if month == February
    call routine2
else if ...
```

The downside of this simple code is that a cracker can easily see all the functions and can examine them one by one. A better way would be to create a function array

```
MYFUN routines[12] = {routine1, routine2, ...};
```

Then, somewhere early in the code, zero out all but the [month-1] index of this array then simply loop through the array until we find the non-zero element and call that one. (We can even hide it better if we manipulate the routine addresses so initially each one has some offset +Mi then adjust the element in the array. Alternatively, the code can collect the array from various data places in multiple code places, and put the routine address into the array only for the current month.)

Make Sure Your Random Seeding is Different at Each Run

To randomize some aspects of the license checking, we need some routines that give us semi-random values. For C programs, that is usually rand() – which is often seeded by a call srand(time(NULL)). The problem with this simplistic randomization is that it is usually done only once at the very beginning of the application. It is easy to find and, if the seeding is patched out, the cracker can ensure that the application is running predictably every time – all our attempted randomizations are patched out, too. Some disassemblers even have C library code detection, so the cracker can see the rand() and srand() calls in the disassembly list.

To counter that, we may seed srand at multiple places; or better yet, create our own rand/srand pairs. Also, it is advisable to seed any such pseudo-random function with something that is harder to find in the code. If we can check that the code runs on a Pentium class computer, than we may use the RDTSC instruction that gives us a very good, random-looking seed value. We may also combine multiple runtime values (time, process id, etc) to seed our srand(). If our own srand() just uses a global seed/next value, than we can manipulate it directly (or, rather, via indirect access) from multiple places of the code.

Create Multiple Private srand / rand Functions

Using the standard srand() and rand() functions has the disadvantage that a cracker may easily find all places where we call them – and that in itself may be a giveaway of the location of the security routines. If we create a multitude of MySrandN() / MyRandN() function pairs, we can make it harder to find all of them. Note that for just the software protection routines we don't really need perfect randomness. In fact, since we just replace the use of rand() – which is cryptographically weak – we need only sufficiently random pseudo-random generator routines. We can start from the usual – and very simple – rand() function

```
int MyRand (void)
{
    return(MyRandSeed = MyRandSeed * 214013L + 2531011L);
}
```

while MySrand() just saves the seed parameter into variable MyRandSeed.

By slightly varying the constant values – just keep them odd numbers – we can create many random number functions.

Please be advised that these (and the standard rand function) should *not* be used in any other code (eg. for authentication, generating session keys, encrypting data etc.). For these types of requirements, you would use a cryptographically good random source. (In Windows you should use CAPI or CAPICOM.)

Checksum Your Code

Since most cracks work by modifying (patching) the code, it is important to make sure that such code modification can be detected. Depending on the environment (Windows vs. Linux), executable type (application vs. DLL) and compiler/linker used (some put imports into the code area), checksumming the code may be easy and straightforward or may need more elaborate code.

The simplest way occurs when the code is not changing (eg. it is a Windows application always loaded at the usual preferred address 0x400000). The code would just need to know where it starts and where it ends – and even that is easy to get from the in-memory executable header. In the case of DLLs, relocations have to be taken into account. If the import function table is in the code section, that area needs to be ignored in the calculation.

It is important to use strong checksumming, preferably some cryptographic hash functions. MD5 and SHA-1 are good for this purpose (their cryptographic weakness is related to pre-generating two different images that would hash to the same value – but we don't deliberately create two code bases. These functions are still good at one-wayness, ie. that it is unfeasible to patch the code that would still hash to the same as ours.)

First, we need to calculate the known good code checksum and store it somewhere. One can write a simple utility to calculate the value then store it either somewhere in the header or in a marked-up data area. When the application is running, it recalculates the checksum and verifies that it is the same as the good, known value.

Of course, a cracker may find the checking routine and patch that out, too. Or recalculate the checksum after the patch and replace the original value with the new one (in the executable image).

It is important to employ some of the other tricks to prevent that. One technique is to calculate the checksum (hash) but compare only part of it – then use the other part in some calculations. We can also calculate some other values based on the hash and store those as data. Now when we recalculate the hash at runtime, we can recalculate some other values based on the just calculated checksum (hash) and the stored other values – then use those somewhere in the application. If a cracker modifies the code, even if he replaces the expected checksum value in the image, these other calculations will result in incorrect values and the application will not work correctly.

Make Changes between Releases

Crackers learn and remember. Once an application is cracked, they know where to look. When the next release comes out, they are likely to try the same patching to crack the new version too. Even if they are unable to fully crack the program, they may still remember how far they reached in analyzing the protection code. If the new release (binary) code is almost the same as the previous one, crackers will have an easier time finding the same functions and data.

It is important to always improve the protection between releases. Even if the application is not cracked (or the software vendor just doesn't know about it) developers should add new code to the software protection area. One may also just change, or re-shuffle, code and put various functions into different source files etc.

Another way to radically change the binary code is to control the link sequence: the order the various objects are linked together. Just by re-shuffling the link sequence code, the data will move around and the code will not look different. This may be improved further by adding various dummy function calls or extra layers between calls, which will prevent even function tree mapping matches. (The latter is an advanced technique some hackers use when trying to figure out changes made by security patches so they can see what was fixed. They then create exploits that work on the older, unpatched code.)

Hide Code with Anti-Disassembler Technique

To make it harder for crackers to analyze the code we can employ two techniques:

- Encrypt parts of the code
- Add anti-disassembler code

The first technique will completely prevent static disassemblers from analyzing the encrypted code part. However, in order to execute, the code needs to be decrypted and at that point it can be disassembled (either within a debugger or dumped out of memory and with a static disassembler).

The second technique adds code bytes that throw the disassembler 'off the track' ie. it will misinterpret the code. This works only with variable-length instructions (eg. Intel x86). The trick is to make sure the actual code flow 'goes around' some bytes that look like the start of a longer instruction. When that byte is encountered by the disassembler, it interprets it as the longer instruction – then the *real* instruction that follows will not be shown in the disassembly list. Depending on the intelligence of the disassembler, there are various ways to achieve this. Note that disassemblers in debuggers are usually unintelligent so it is easy to fool them. Some static disassemblers, on the other hand, are clever enough to analyze deeper and see if the phony byte is unreachable – or that a jump indicates code flow after that byte. The surest way is to use a call which simply increments the return address to skip the following byte.

Anti-disassembly code:

```
__asm call SkipNextByte
__asm __emit 0xe8
... code to hide here ...
```

The function:

```
void SkipNextByte(void)
{
    __asm inc dword ptr [esp]
}
```

Of course, if we want to do this in a large number of places we can create various macros that make it easier to insert. One should vary not only the dummy code byte, but also the anti-disassembly code style. An anti-disassembly macro code generator can also be developed which can generate large numbers of randomized macros so it would be hard to find and wipe them out. Such a generator may also help in changing the generated code between releases – one would just generate new macros for each release build. Such anti-disassembler randomized code can be automatically generated with some software protection tools.



Check for Debuggers

Crackers usually need to debug a protected application to follow the code flow and understand where license checking occurs. To protect their application, developers can detect when it is debugged, and, if so, terminate the program. There are multiple ways to detect debuggers.

Under Windows, the easiest way is to call the `IsDebuggerPresent` API and, if it returns `TRUE`, then the application is being debugged. Of course, this is the least reliable method as crackers may monitor this call and, if encountered, change the return value to `FALSE`. Under NT in the TIB (Thread Information Block) there is a flag that shows the same so the code can directly check this. System-level debuggers (eg. SoftICE, KD) are not shown by this function.

With the help of SEH (Structured Exception Handling in Windows) one can issue debug interrupts (INT 1 / INT 3). Usually if a debugger is present, it will break into the debugger – while if there is no debugger, it causes an exception that the SEH handler can detect. So the check is simple: if we don't get the exception in the SEH routine, there is a debugger. A more elaborate way to use SEH is to set and use the Intel x86 processor's debug registers. Again, if these are used by the debugger, then we can detect that (as our code will not be able to use them).

SoftICE is a particularly good system level debugger preferred by many crackers. In addition to the above mentioned INT 1 / INT 3 and debug register use detection methods, there are various ways to detect if it is running, such as by using some interrupts, checking device driver handle, and checking service. (Note that checking for SoftICE is different on Win9x and NT platforms.)

Expert crackers may know some of these checks and, by looking for them, may be able to skip the code that checks debugger presence. However, by using SEH developers can still detect such bypassed code – just add some code to the exception handling routine which depends on what exceptions, and from where, they were derived. calculating some values based on them – then later check that these values are correct (or even better, use those calculated values). If the cracker skips the debugger checks, than our SEH routine will not be called and we can find that out later.

Best-Known Debugging / Cracking Tools

Debuggers:

Microsoft WinDbg, Visual Studio's built-in debugger
Compuware SoftICE

OllyDbg - free download at

http://downloads-zdnet.com.com/OllyDbg/3000-2383_2-10242634.html

Disassemblers:

IDA Pro - see

<http://www.datarescue.com>

Monitoring tools:

SysInternals FileMon, RegMon, PortMon, PMon, Process Explorer - free downloads at

<http://www.sysinternals.com/Utilities/Filemon.html>

<http://www.sysinternals.com/Utilities/Regmon.html>

<http://www.sysinternals.com/Utilities/Portmon.html>

<http://www.sysinternals.com/Utilities/Pmon.html>

<http://www.sysinternals.com/Utilities/ProcessExplorer.html>

Executable viewers:

Microsoft Process Viewer and other Visual Studio tools

PC Magazine utility FileSnoop - see

<http://www.pcmag.com/article2/0,1895,1570739,00.asp>

QuickView Plus - see

<http://cws.internet.com/file/11510.htm>

Summary

Use a Skilled Software Security Vendor

Draw upon the expertise of an organization dedicated to protecting software.

Do Not Check Only for the Presence of a License

Good protection code must ensure that it is insufficient to simply patch the code to remove a call to check for the presence of a license at start up.

Hide Calls

By obtaining the function addresses at runtime, they will not be listed in the import table and therefore will be much more difficult for a hacker to find after disassembling the code.

Check If External Functions Have Breakpoints

Check if somebody is trying to monitor Windows APIs used by the protection code.

Hide Strings

Even if one employs the trick to get function addresses directly from DLLs, the function names still need to be checked.

Don't Use the Same Functions Everywhere

If the most commonly used protection function is patched out, it does not matter how often the program tries to check the license.

Spread Out Checking Routines in the Code Section

Effective security requires spreading the software protection functions all over the code base.

Use Silent Code

Important security code parts should be put into such silent code blocks, then these code areas should be put into "normal" application (ie. not security related) code.

Calculate Results – Then Use Them

Instead of a simple comparison, the code should receive a response then, much later, use the obtained value for some important internal calculations.

Check Part of a Result – Then Crash if the Other Part is Invalid

A cracker may find a compare place and patch it out, but the code could later use the other half of the result to do some calculations and derive some important constants.

Hide Access to Variables (via Array Indirect Access)

You may have data that is accessed from various functions and used later somewhere else in a calculation. We need to hide *where* we access the data.

Access Variables via Allocated Memory

Even if we re-write the code so the code will not show the address of data we use a cracker can use memory breakpoints in a debugger to find out where we use the data.

Copy Variables to Multiple Places

One debugging tool crackers often use is the Intel x86 processor's debug registers. However, there is one limitation of this tool that can be used to the developer's advantage: use many more variables so the cracker cannot track them all.

Access Global Variables Often

Another way to make tracking variable use difficult is to simply use them too often.

Use Macros – Generously Sprinkle

Through the use of macros, we can access the variables in many places throughout the application code without the need to add the same code everywhere manually.

Allocate Random Amount of Memory at Startup

Allocating a random amount of memory at the very beginning of the application is a simple way to make sure at least some data addresses are different each time.

Create Variations of Checking Code

Use multiple ways to check the license condition.

Make Sure Your Random Seeding is Different at Each Run

Use some routines that give semi-random values in order to randomize some aspects of the license checking.

Create Multiple Private srand / rand Functions

Creating a multitude of MySrandN() / MyRandN() function pairs makes it harder to find all of them.

Checksum Your Code

Since most cracks work by modifying (patching) the code, it is important to make sure that such code modification can be detected.

Make Changes between Releases

If the new release (binary) code is almost the same as the previous one, crackers will have an easier time finding the same functions and data. It is important to always improve the protection between releases

Hide Code with Anti-Disassembler Technique

To make it harder for crackers to analyze the code, we can employ techniques prevent static disassemblers from analyzing the encrypted code part and techniques that add code bytes that cause the disassembler to misinterpret the code.

Check for Debuggers

You can detect when an application is debugged, and, if yes, terminate the program.



SafeNet Overview

SafeNet is a global leader in information security. Founded more than 20 years ago, the company provides complete security utilizing its encryption technologies to protect communications, intellectual property, and digital identities, and offers a full spectrum of products, including hardware, software, and chips. UBS, Nokia, Fujitsu, Hitachi, Bank of America, Adobe, Cisco Systems, Microsoft, Samsung, Texas Instruments, the U.S. Departments of Defense and Homeland Security, the U.S. Internal Revenue Service, and scores of other customers entrust their security needs to SafeNet. SafeNet was taken private by Vector Capital in 2007.

For more information, visit www.safenet-inc.com.

Corporate Headquarters

4690 Millennium Drive, Belcamp, Maryland 21017 USA

Tel.: +1 410 931 7500 or 800 533 3958, Fax: +1 410 931 7524,

Email: info@safenet-inc.com

EMEA Headquarters

Tel.: + 44 (0) 1276 608 000

Email: info.emea@safenet-inc.com

APAC Headquarters

Tel: +852 3157 7111

Email: info.apac@safenet-inc.com

For all office locations and contact information, please visit www.safenet-inc.com/company/contact.asp

©2008 SafeNet, Inc. All rights reserved. SafeNet and SafeNet logo are registered trademarks of SafeNet. All other product names are trademarks of their respective owners.

