Toughening Software Protections

By

Sabuj Pattanayek

V1.0 April 6th, 2002

V1.1 April 21st, 2002

V1.2 June 13th, 2002

Background

Although this paper would have been ultimately written, much impetus was provided by the Federal Government's (U. S. Custom Department's) investigation into Drink Or Die (DOD), the "underground" software securities group. The following critique on software licensing systems given in this essay is meant to increase the effectiveness of already available software protections by shedding light on their vulnerabilities and strengths. This paper will also attempt to demonstrate how software hackers think, what tools they use, why they attack certain targets, and how to keep software from being "cracked". It is hoped that software authors and protectionists will take the given recommendations seriously (or at least attempt them experimentally to some degree).

The Internet has created a medium by which software authors can advertise their evaluation, shareware, and demo software applications for users to download and examine. By software applications I mean everything *but* games and visually stimulating interactive software[1]. Websites such as ZDNET's HotFiles (http://www.hotfiles.com) and CNET's Download.Com (http://www.download.com), allow authors to post easily such software according to utility. Alternatively, authors post download links of their evaluation software on their own Internet web pages and public FTP[2] sites. The majority of these titles allow a certain level of licensed evaluation. For example, a software may

---

[1] Interactive games and simulations on digital versatile disks (DVD's).

[2] Users can anonymously login to public FTP or file transfer protocol sites to download evaluation software.

allow a user to operate the software for a certain time (e.g. five minutes or thirty days), or may only allow a certain number of runs (the software may be executed a maximum of ten times), or may disable functions such as saving and printing, or may simply display nagging messages reminding the user to buy the software. These restrictions are lifted when the user fully licenses the software.

Software licensing is done through a variety of means ranging from relatively simple registration or serial codes to more elaborate (not necessarily more effective) key files[3] and dongles[4]. Hackers and software authors themselves, soon discovered that they could debug, disassemble, and reverse engineer these security mechanisms. This would allow them to find vulnerabilities and loopholes, ultimately leading to the emulation of any license required by the software. These people became known as "crackers", a term (and its derivatives) that I will no longer use because of its inherent negative connotations. Information regarding licensing mechanisms held only by software authors and debugging techniques known only to programmers adept in machine languages[5] was disseminated by the rapid growth of the Internet. Hypothetically speaking, the information[6] and the tools[7] are available today to anyone who is patient, has a curious

---

[3] Usually a binary or text file which unlocks certain features in the software.

[4] A hardware device usually connected to the parallel port (printer port) or USB (universal serial bus) port of a computer that communicates licensing information with the program, thus unlocking certain features of the program.

[5] All CPU's (central processing units) at "low level" execute some form of Assembly language or machine language. The compilers of high level (object oriented) languages such as Pascal, Basic, and C++ all translate their respective languages into the Assembly language understood by the CPU.

mind, and an AOL (America Online) account.

What will follow is a critique of the various types of licensing schemes and the software protections which secure (or claim to secure) the licensing mechanisms. If the narrative has been difficult to follow or understand thus far, it will become more technically involved, and thus difficult reading for anyone who does not possess some knowledge of computer programming. Although the majority of this text will not reference directly much computer code, it is assumed that a programmer interested in creating better protections will seek out the code by which the security methods mentioned may be implemented.

What tools do software hackers and reverse engineers use?

There are various debuggers and disassemblers available that allow reverse engineering of executable and non – executable code. I will cover mainly the ones that I am familiar with. These are Compuware Numega SoftICE (available with Compuware DriverStudio and Driver Suite), URSoftware W32Dasm, Datarescue Interactive Disassembler Pro (IDA), Oleh Yuschuk's OllyDbg, and Eugene Suslikov's Hiew. All of these serve a similar purpose, that is to display the low level x86 assembly language of

---

[6] For examples of such texts see http://www.google.com/search?hl=en&safe=off&q=how+to+remove+software+protections.

[7] Programming and debugging tools (programs which disassemble the target software's compiled executable code into Assembly language both interactively at runtime and also when not being executed) are similarly available by querying search engines: http://www.google.com/search?hl=en&safe=off&q=debug+program+reverse+engineer+tools.

the target code on all Microsoft based operating systems, either in real time or after disassembly has been completed.

I. The software and computer industry's premier debugging tool for Windows based machines is SoftICE. Originally intended and still used for the development of Windows device drivers, it has become a double - edged sword that has allowed hackers and programmers alike to analyze and modify code in real time (both in memory and on disk) without having to recompile the object code. I would have a difficult time believing that the geniuses who developed this software did not realize the power it would give to curious individuals. The program has become so ubiquitous and abused that many authors have included anti – SoftICE code in their software[8]. In turn, hackers have created programs such as IcePatch[9] and FrogsICE[10] to protect their valuable asset from being detected and their goals from being thwarted. I will discuss how to best implement and use debugger detection to ward off hackers in the "General Strategies" section.

Most hackers who regularly use SoftICE have an extensive knowledge of Windows core API (Application Programming Interface) functions. These functions are located in two main DLL's (Dynamic Linking Libraries) that are at the heart of the Windows operating system, KERNEL32.DLL (KERNEL.DLL is the 16-bit counterpart) and USER32.DLL. All development platforms included with Microsoft's Visual Studio (and Visual Studio .NET versions) platforms use imported functions from these DLL's to do tasks like creating windows. SoftICE has the ability to load any DLL or other function library and then read its function exports section. This gives the user the power to set a breakpoint (using the BPX command in SoftICE) on any function or address in

---

[8] See http://linux20368.dn.net/crackz/Tutorials/Protect.htm for detailed methods on how to code against the use of SoftICE.

[9] Located at http://linux20368.dn.net/protools/files/debuggers/icepatch.zip.

[10] Located at http://linux20368.dn.net/protools/files/debuggers/frogsice.zip.

memory which he believes is being called by the target program. For example, the function ShowWindow is almost always called when creating, drawing, or re – drawing an object on the screen. If a nagging message such as a "Trial expired" window is constantly appearing, it is possible to set a breakpoint in SoftICE by doing "BPX ShowWindow". When the function is called by the program, SoftICE will appear and the user may begin tracing through the assembly code. It is easy to see the implications of this, when one realizes that many software protections (such as FLEXlm and Wibu – KEY) use functions embedded in external DLL's. Thus, it is always a good idea to statically link or embed DLL files that are worth hiding directly into the executable code.

SoftICE can also be used to find string references within object code. Once found, it returns the address in memory. Continuing with our "Trial Expired" example, it is possible to search for this string in the object code by giving the command "s 0 l ffffffff 'Trial Expired'". Then the user can place a "break on memory access" on the memory location returned from the string search by using the "BPM" and "BPR" commands. When the program attempts to access the "Trial Expired" string reference in memory SoftICE will be triggered.

Alternatively a user can use the command "HWND" to list the current window and other on – screen object handles in memory. Then by using the "BMSG" command, a user can place a breakpoint on a handle reference and SoftICE will be triggered when that object is accessed. One can see how useful this may be in tracking down where in the object code certain object handles are referred.

Additionally, access attempts to parallel ports used by dongles and other external hardware locking devices can be monitored through the use of the "BPIO" command to trap calls to IRQ ports. Access to system interrupts, such as those used by the keyboard, CD-ROM, and other drives can be trapped using the "BPINT" command. Keystrokes can be intercepted before the target program receives them.

II. IDA Pro is perhaps the most highly configurable and most complex

disassembler on the market. One of the key features of this program that has been abused by hackers is its ability to apply FLIRT "signature files" to a disassembly. FLIRT "sigs" are used to identify functions within the disassembly. One legitimate purpose of this may be to locate statically linked functions from DLL's used by Visual C++ (MSVCRT.DLL and MFC*.DLL). Hackers have instead created "sig" files for various versions of FLEXlm, Rainbow Sentinel dongles, and HASP dongles. This allows the hacker to locate the hexadecimal address of certain key functions such as the Sentinel SuperPro dongle's initialization function "sproFindFirstUnit" and FLEXlm's "lc_init" and "lc_checkout" functions. The code at the address of interest can be replaced with an "INT 3" assembly "opcode". Then, in SoftICE a "BPINT 3" command can be issued to intercept the "INT 3" opcode upon execution. Since IDA is quite meticulous in its disassembly, it is also perhaps the slowest disassembler. One alternative will be discussed next.

III. W32Dasm is a limited, yet quick alternative to IDA Pro. It has the ability to easily display references to all imported and exported functions in the object code of interest. It can also display all string references and dialog references used as resources in the object code. Locating a string reference such as "Trial Expired" can be as simple as double clicking on the string in the W32Dasm string references window. The assembly code which accesses the string is instantly shown. It is good practice to not hardcode text strings into the object code. Instead the string "Trial Expired" can be dynamically created at program execution.

W32Dasm also has a built in real – time debugger which is more user and beginner friendly than SoftICE. It also includes an API reference manual of sorts, that displays function prototypes whenever core system functions such as those from KERNEL32 and USER32 are called while W32Dasm is in debugging mode. W32Dasm has become so popular with hackers that several modifications for this program have

been created to make it even more powerful and easy to use[11].

IV. OllyDbg unlike SoftICE is a debugger that does not operate in Ring 0, that is, it does not situate itself between the operating system and the main CPU (this is why SoftICE loads itself into memory before Windows initializes), but rather runs directly as a Windows 32 – bit (Win32) executable. This is one of the most recent debuggers that have been created for Windows systems. It has many of the same functions and abilities as SoftICE, but its interface can be intimidating. OllyDbg has become popular for reverse engineers working with Windows XP, since even the latest version of SoftICE included with DriverStudio v2.6 has been reported as having troubles with Microsoft's newest operating system. OllyDbg is also a good alternative for many users since it is available free of charge.

V. Hiew, short for "Hackers View", is still being developed by its author. It is a hex - editor, binary file viewer, disassembler and assembler all in one. But unlike IDA and W32Dasm it does not fully display function references. By disregarding this option, the author has created a program which gives an instantaneous and "dirty" disassembly. This program is mainly used by hackers in conjunction with more powerful disassemblers and debuggers. Once the user knows what and where to modify in the object code, it is simply a matter of loading up the file in Hiew and making the modifications directly to disk. It is more convenient to use than a simple hex – editor since it has the ability to show the equivalent disassembly for the given hex code.

VI. There are also many other forensics and analysis tools used by reverse engineers from time to time. Filemon[12] allows a user to see all the files being accessed by the

---

[11] See top of page at http://linux20368.dn.net/protools/decompilers.htm.

[12] Available from http://www.sysinternals.com/ntw2k/source/filemon.shtml.

operating system as well as by running programs. This is often used to locate license keyfiles and other data files that are accessed by programs. Regmon, also created by SysInternals allows a user to view how the system and programs running on the system access the Windows system registry. Often license keys, RSA encryption seeds, and other pertinent information to a program, such as the date that it was installed (and how many days have passed since install) are kept in the registry.

CodeFusion[13] is a patch generation engine that is used by many hackers to create executable patch files that can modify other target files. Patches can be created in various ways. One method is by giving CodeFusion the hex addresses, and the bytes to be changed at those addresses. Another method is by file comparison, where the original target file(s) is compared with the altered file(s). In either case the location and the bytes to be changed at that location are stored in the patch file. The created patch file is usually very small (less than 50 kilobytes), and gives the recipient of the patch the ability to discover how the target files are altered by the patch, and what effect those changes can have on the program. More than 95% of DrinkOrDie "releases" included an executable patch file and the original unaltered demo, shareware, or evaluation program setup files.

ProcDump[14] is a "memory dumper" that can write the contents of any process or file loaded in memory to disk. This method of analysis is often used by hackers against programs that are packed and or encrypted prior to being executed. That is, before a packed or encrypted program can execute, it must usually unpack and or un – encrypt itself. The unpacked and un – encrypted "virgin" product is loaded into memory where it can be written to disk using a program such as ProcDump. ProcDump also comes built in with several scripted unwrappers for commercial packing schemes such as ASPack, Armadillo, TELock, and older versions of the popular commercial protection scheme VBox. In addition ProcDump can be used to edit the PE header of Win32 executables.

---

[13] Located at http://my.magicpage.co.il/Comp/kobik/download/codefs30.zip.

[14] See top of page at http://linux20368.dn.net/protools/unpackers.htm.

> IonWorx Software[15] has realized the threat these tools can pose to protections, and has created modules for Delphi and C++ Builder development platforms that can be integrated into a program prior to compilation. These modules contain code to crash debuggers, disassemblers, and system monitoring programs such as those discussed above.

There are various options and tools[16] a reverse engineer can use to analyze a target. New tools are created quite often, mostly to "un – do" the packing or encryption done by commercial protections schemes. One such recent example is Tsehp's Revirgin[17]. This small utility was created for the sole purpose of rebuilding trashed or damaged IAT's (import allocation tables) created as the result of applying commercially available wrappers such as VBox and CDillaLM to executable code. Without an intact IAT, a Win32 executable will always crash when attempting to call a core Windows function or any other imported function reference (which it does quite regularly). To a large extent, Tsehp's tool has automated a task which would have otherwise taken a reverse engineer hours, if not days of work.

This is just one example of how the software protection game is further propagated. I would be obliged to say that an entire industry has been created based on this game. New exploits are found, many of which I have already delineated. Soon there comes a need to patch the holes or create a new protection scheme. The industry grows due to the new technology, new jobs are made, and new ways of thinking about the

---

15 See http://www.ionworx.com/ADPII.htm.

16 See http://protools.cjb.net.

17 Available at http://www.woodmann.com/fravia/exe/revirgin.zip.

problem is the result. Then a reverse engineer or hacker comes along, defeats the protection, and the cycle restarts.

1) Q: How do I prevent the licensing system from being compromised?

A: Do not provide a licensing system in downloadable demos, evaluations, and sharewares. Do not provide a target.

As trivial as the following suggestion may seem, it will greatly prevent the majority of software protections and thus software titles from being exploited. It will go a long way to ending the software protection game that exists between hackers and authors, and put more blame on the true pirates who simply steal and duplicate software with included licenses. I still do not understand why the majority of software authors make available for download versions of their software with all of the program's functionalities intact (within the executable code) albeit crippled to some extent. That is, authors find it necessary to provide versions that are time limited, limited by number of runs, and limited by crippling functions that still exist within the program code but are blocked off behind a "locked door". Nowadays it is much easier for an individual to find a patch or a key – generator for shareware and demos on the Internet[18] than a fully pirated program. How does one prevent such software from being exploited?

I. Do not offer software for evaluation purposes if it includes all the code and thus

[18] See http://www.astalavista.com.

all the functionality the fully licensed version includes. If the "Save" option under the "File" menu is "grayed out", make sure that "ungraying" it will have no effect. The demo version must be truly crippled in that it does not include necessary code to execute a certain feature (e.g. saving). Then there is no reason why the author should go to any lengths at all to provide a licensing system (unless it is simply to distract the hacker into thinking that there is actually a target or something behind the "locked door").

II. Relativity theory tells us that time is not absolute, thus the software author should realize that the passage of time on a computer can be emulated or falsified. With very few exceptions (that will be discussed in section 5) time limiting full versions of software titles provides the easiest route for a hacker to enable the program for an indefinite amount of time. If you must provide time limited software, make sure it is fully crippled in some manner, and that notification of being time limited is only provided once (e.g. during installation). Then leave it up to the hacker to figure out why the program is not starting after the time limit has expired. For example, do not provide audible (as in beeps) or visual notification (as in nagging pop-up windows) that the evaluation time limit will expire in a given amount of time or that it has expired.

In summary, do not give the hacker a target. Make the hacker think there is nothing to attain behind some "locked door" (even if there is). Always remember that advanced reverse engineers (I will refer to as "RE" or plural "RE's" henceforth) will nonetheless fully disassemble the object code themselves and conduct their own investigation of what the author's software includes and does not include, regardless of what the author may claim. So it is a good idea to always distribute truly crippled demos and shareware versions.

Sometimes authors who only provide evaluation versions on request (as in high

end engineering software applications useful to a computing minority), that is versions not usually available for download from their websites, leave full evaluation versions or full commercial versions lounging around on their anonymous login company FTP sites. While this does provide easy access for customers to pick up version updates and upgrades, it also provides the hacker a target software application. It would be much safer to provide each customer with individualized logins and passwords. Access to the FTP server is minimized and FTP access logs are much easier to analyze.

2) Q: I need a licensing system or software protection for my software. What are the options? What are their strengths and weaknesses?

The majority of software available today are "generic" duplicates of "name brand" software created by large corporations (Adobe, Autodesk, Symantec, etc) who have arsenals of programmers working for them. Software hackers usually do not care for such "generic" software (or most software at all) and attack any protection simply because it is exists. Usually hackers will attempt a protection simply because it may provide a challenge (or an alternative to boredom), like a crossword puzzle. I suggest the author spend more time coding something novel into the features of the program itself. Then if there is really something worth protecting, the following guidelines may be of assistance.

I highly recommend that you do not use most commercial software protections or licensing schemes. If a hacker is able to compromise a single commercial protection, then all software titles protected or licensed using that mechanism have been effectively

compromised. However, the main problem with using any commercial protection scheme is that the company selling the protection rarely provides any in depth information about how the protection really operates. Does it use simple validation functions or does it integrate itself more with the software? The author is left to blindly implement the protection using only the specified instructions given to them by the protection system's documents. The hacker can access these documents[19] just as easily as the author and understand the protection's implementation. Failure of the protection almost always occurs because of the implementation of the security. Imagine for example, a steel door attached to a wall made out of paper. For the hacker it is much easier to circumvent the door by making a passage through the paper wall. In this case, the security was poorly implemented. I will now critique several licensing protection systems based on my own experiences of reverse engineering the various schemes.

3) Dongles (External Hardware Locking Devices)

[19] Implementation of most security systems is done through API calls to DLL's or other library files. The company provides instructions on which API functions to call and what order to execute them in. Even if these documents are not available to the hacker, it is possible to simply disassemble the DLL and then look at its exported API functions. More and more commercial protection building software today claim to offer simple drag and drop installation of the protection into any executable code. Some of these protections include Aladdin's Vbox (created by Preview Systems) and several protection solutions offered by Bit-Arts. These programs offer strong protections in that they provide the author (and thus the hacker) with little knowledge of how the target code is being protected. Protections such as Vbox and Bit-Arts solutions will be discussed in section 5.

The basic premise behind a dongle or any other external hardware licensing mechanism is that it communicates codes and license information with the software when the application requests such information. The majority of applications downloadable from the internet that are protected with dongles revert to some crippled demo version of the software or simply do not launch, usually notifying the user with a nagging window message when the program does not detect the dongle. As explained in the earlier section, there would be no need for a dongle licensing system if the demo version were already truly crippled.

Most dongles in use today are created by Rainbow Technologies (Sentinel, SentinelSuperPro), Aladdin Knowledge Systems (Hardlock, HASP, MemoHASP, TimeHASP, etc), and Wibu Systems (Wibu - Key). These are the weaknesses of these systems:

I. The company will usually provide software development kits (SDK's) for programming the license systems into the target code. These give software authors and hackers, information on the licensing system's API functions. The company literally provides the hacker with the internals of the security system! This would be similar to ADT (a leader in private home security systems) giving the internals of their mechanisms to anyone interested, and then selling the security system to customers. Even if the author directly embeds the protection into his target code (without referencing external DLL's), the hacker knows what signatures to look for (e.g. a certain hexadecimal string or certain variables passed to key functions of the protection system). Calls to Sentinel and SentinelSuperPro (SSPro), HASP (all variants), and Wibu - KEY dongles are easy to locate within a program which implement these devices. It only becomes a matter of time before missing pieces of the puzzle are figured out.

II. The software must query the dongle and then check to make sure the returned values are correct. The hacker can easily reverse engineer validation algorithms within the software and emulate the proper dongle return codes. The main weakness is that fabricating and programming dongles specific for a particular application can be very time consuming and expensive, thus most software titles retain the same dongle (which returns the same codes) through various versions of the program. If the hacker acquires the dongle itself, its memory can be fully explored (dumped), emulated, and eventually all return codes known. The dongle then becomes obsolete.

I do not recommend working with dongle systems unless customers are willing to absorb the costs necessary to implement the dongle. If a dongle licensing system has already been invested in and is being used by customers, chances are it has been compromised. The following characteristics are strengths of dongle systems and implementation suggestions to better the systems:

I. Do not use validation functions which return simple values such as 1 (dongle OK) or 0 (dongle not present). This is a completely incorrect approach because it assumes the dongle is returning serial numbers or codes required to “unlock doors”. Remember that there are no “doors” in software. If the hacker can see some of the software code, it usually means all of it can be seen and analyzed (unless the software incorporates self modifying code at runtime, polymorphism, or encrypted or packed code…this will be explained later in this section).

The approach should be to somehow integrate the dongle memory with its program. Most dongles have large memory areas that can be used to store program instructions necessary for the operation of the software. For example leave out certain program instructions (program code) necessary for the software to operate. There is

nothing more irritating or diffusing for a hacker than a crashing program. Most hackers will simply give up believing that the author forgot to weed out bugs in his own program.

After receiving the memory from any dongle (or from inputs entered by the hacker directly into program memory at runtime), do not immediately validate them, do not create nag messages notifying the hacker anything is wrong. Create a hash from the data value, chop it up into smaller bytes and scatter it in memory. Use the scattered hash values as indices for arrays or data structures. If this is properly done (the dongle returns correct values), then the data structures should be ok, but if improperly done (hacked or dongle returns no values) the data structures can be mangled (or should stay mangled). This may ultimately cause the program to crash at runtime or return strange values from vital program functions (making it useless). Dongle query results can be used to return program execution addresses (EIP hexadecimal register values, e.g. 0040A73Fh) in small bytes. The addresses, once recompiled byte by byte, can be used to control which functions the program calls and how the program is executed (or whether it crashes). This creates several challenges for the hacker. First the hacker must somehow keep track of all this dongle data floating about in memory, and then trace how, when, and where the data is recompiled, and finally trace at what location(s) the software uses this data to decide execution paths, and finally what the software should attempt to execute as the next step. To make things even more challenging these addresses can be encrypted within dongle memory and then decrypted at runtime using other values returned from dongle queries, key files, windows registry, or any other external location.

In another example of integrating license data, some CNC (computer numerical control), CAD (computer aided design), CAM (computer aided modeling) software use licensing information to directly determine the machining and cutting process. Incorrect license information can be used to calculate incorrect machining paths. If the software is a 3D graphics program, incorrect information can be used to render images improperly. This can be accomplished by encrypting mathematical constants into the dongle memory or creating tables of encrypted variables to be passed to functions (referred to as the

stack). Once again, direct "yes or no" validation routines should not be used in these processes, but somehow the license information should be intimately tied into making crucial calculations. All of this creates a "tight rope" for the hacker, yet one that does not follow a linear and straight path.

II. Alter the dongle memory with each successive major version of the program. Offer dongles with different memories to different customers. Do not create a universal dongle for the software. This is like creating a hard coded serial number and giving it to all customers. However, if the dongle memory is successfully integrated into the program and incorporates the dongle data in many aspects of the software, it may be difficult and time consuming to rewrite all the algorithms for multiple dongles with different memories. In this case a more modifiable variable may be used in addition to the dongle, such as an additional licensing system which uses encrypted data or key files created for various levels of licensing. Details on key files, encrypted serial numbers, and how to use them with dongles will be explained in section 6.

III. Newer versions of the HASP API implement HASP objects which incorporate self modifying code (SMC). SMC itself is a nightmare to trace through when debugging. The HASP object SMC also contains many unconditional jumps which can easily frustrate any experienced RE. The code becomes very confusing because it seems to lose its causal and deterministic behavior.

IV. The generic "out of the box" implementation for the Hardlock dongle is by far the most effective. It packs and encrypts the executable object code (making it impossible to disassemble the code), destroys its import allocation table, and has built in debugger detection[20]. Without the dongle itself it is nearly impossible to recreate the

[20] Debugger detection is implemented into many commercial protections. Usually once the program

import allocation table, rendering a useless and crash prone software title.

4) Complex Commercial Licensing Schemes

These schemes are complex in that they provide a wide variety of licensing options and licensing levels for the author to easily integrate into the software. They are however the most weak and easily exploited of all protections since the internals of the licensing schemes are well documented. These include GLOBEtrotter (FLEXlm), Rainbow Technologies (SentinelLM and ElanLM). I highly advise against the use of any of these protections. The downfall of all these server based licensing systems is that they are all built on “yes or no” validation functions that are easy to discover and alter into providing correct return values. Out of the box implementations offer no CRC checking of the object code to test whether it has been altered. Here are several more weaknesses:

I. Since the internals of the FLEXlm licensing system are quite well documented, they can be altered to provide any level of licensing. The new version 8 of FLEXlm implementing ECC (elliptical curve cryptography) are no better than the original versions. They simply include more encrypted keys and seeds, some of which cannot be easily recovered from the FLEXlm data structure without brute forcing. However for the RE, acquiring keys and seeds is only necessary to recreate original license key files. In this case it is much easier to go around the steel door than to go through it. It does not

---

detects active debuggers, such as Numega's SoftIce, TRW2000, etc the program notifies the user and ceases execution. One good technique to use with debugger detection is diversion. The message displayed by the Hardlock wrapper is something like "Hardlock not found" (it's been a while since I've tinkered with one) when in actuality it should say "Debugger detected, don't even think about it!".

take long to emulate the license within the software code itself because the validation functions (e.g. lc_init and lc_checkout) used in the older versions of FLEXlm still exist in this newer version in one form or another.

Several authors have attempted using some tricks to bolster FLEXlm. Some have attempted to encrypt "FEATURE" names. These are the names of the "licenseable" features of the software found in the FLEXlm license text files. The problem is that the names must be decrypted before passing them to the main licensing function (lc_checkout), creating the opportunity for the RE to "intercept" the actual feature name, and ultimately create a license file. Others have used non windows API's to read the license files. This makes it more difficult for the hacker to place debugging breakpoints on windows API's that read files (e.g. CreateFileA). While manually reading the file, the author checks to see if valid feature names exist, if they do not, the program calls the lc_checkout function passing it a bogus feature name. Otherwise, the program decrypts the valid feature names and passes them to the validation function. The drawback here is that the hacker could acquire an "evaluation" license file from the company, write down the feature names, and then acquire the remaining features not included with the evaluation license by checking the variables passed to the validation function. In an extreme case, the RE could go around digging in the code and ultimately find the feature comparison algorithm without any external reference license files.

II. SentinelLM and ElanLM licensing schemes should also never be used. Both implementations generally use API's linked through DLL files. The DLL's can be directly altered to return correct values since both SentinelLM and ElanLM use simple validation functions. Even if the author were to statically link the DLL into the program's object code (thereby ridding the hacker of the opportunity to intercept DLL calls), the functions themselves remain the same, as do the data structures that are pushed onto the stack before the function is called. The hacker can easily look for these cues as notification of finding the function of interest. Otherwise, a supremely lazy RE can use

IDA Pro signature files in the object code disassembly to locate the functions of interest.

5) "Your evaluation has expired. Please buy this software."

Commercial evaluation and trialware protection schemes have progressed since the early days of Preview Software's TimeLock. Today they are much stronger, impenetrable to the casual and intermediate hacker, and are updated almost monthly. Most of these companies cater to low end software companies because of the feasibility of their protections. These protections include CDillaLM, Crypkey, Vbox, and several Bit - Arts solutions. Why have these protections progressed? Why and how are safer cars created? Hackers stress test protections (usually unofficially) and unfortunately the results of these tests are distributed. Luckily, many hackers that conduct their crash testing experiments ultimately end up working for the protection companies themselves.

Most of these "plug and play" protections offer wrapping of executables and direct injection of the protection into PE (portable executable format) sections. This gives the software author no clue as to how the protection has been automatically implemented, and thus it does not so easily avail itself to the RE. If I were a software author investing in a protection system for program, and knew nothing about coding protections myself, then I would use one of the protections mentioned in this section.

These protections use all the tricks in the book: executable wrapping and packaging, import allocation table destruction, anti - debugging, anti - disassembly, anti - memory dumping, SMC (which usually ends up doing nothing and is simply a distraction), mind boggling amounts of CRC checking (both in memory and physically

on the disk to check whether crucial files have been altered), self regenerating code, hiding of crucial files everywhere on disk[21] and encryption of data into the Windows system registry, and most importantly perhaps more than 50% of the protection's code is not based on Windows Kernel API, but is rather created using raw 32 bit assembly. What does all this do? It prevents the hacker from easily analyzing the protection code and from circumventing the steel door by simply going around it. The hacker is forced to understand, to master the intricacies of the locking mechanism itself. It is like attempting a crossword puzzle written on an amorphous, asymmetrical three dimensional object. The RE has to analyze the protection from all angles. This can be frustrating for the "group" oriented hacker who competes with other groups to be able to circumvent as many protections as possible. However, true hackers will stick with a challenge because this is what they live for, and like any other protection, vulnerabilities may be discovered. Information spreads like wildfires on the Internet. If the protection is compromised, all software protected using the same version of the protection are at risk.

6) General Strategies

The following section is for those who are brave enough to try their hand at

[21] If a program is installed and its evaluation period ends, yet the user desperately wants to use it for a longer period without paying, usually a total reformat of the hard drive will suffice in wiping out any data the protection uses as a reference. CdillaLM however hides itself in the MBR, or master boot record (thus a regular format has no effect). The user will have a surprise in store when attempting to run a reinstalled copy of the program in question. However, running FDISK /MBR will clean the MBR and return it to its original state.

creating their own protection scheme. If the time and interest are available this is the best way to go. A protection made from scratch ensures that only the author knows its true weaknesses, and if it is a strong licensing system, the hacker will have little or no prior knowledge of what to expect or how to go about diffusing the system. Many of the strategies explained in the section on how to implement better dongle protections and even those used in Vbox, Crypkey, etc can and should be used whenever possible. The following are some general strategies and summaries of basic ideas developed in this paper that may be implemented in a protection.

I. The name of the game is not to create a simple door with a locking mechanism, rather the protection must be fully integrated with as many aspects of the program as possible. Use the license data as indices for arrays, linked lists (in mangling data), or for creating execution addresses, anything that may crash the program if improper licensing data is received.

II. A key file can be used like a dongle, except a key file has an unlimited amount of storage that can be exploited. The key file should be used as a data file that can be used to store vital runtime program code. All the dongle implementation tricks discussed earlier can be applied. In fact the methods can be applied to any set of data. Do not give the key file a noticeable name such as LICENSE.DAT, hide the license data somewhere in an inconspicuous DLL file. Encrypt the data so it does not look like serial numbers or codes.

A key file used in conjunction with a dongle bolsters the total security mechanism. Integrate data from the dongle and from the key file. Use one set of data to decrypt the other or vice versa. Both sets of data can then be used in actually operating the program (and not just for validating the license data). Create unique key files for

each customer.

On top of this, another unique key or data set[22] may be used. Now there are two unique sets of licensing data for each customer (the data in the file and the data the customer has to directly enter into the program) and there is also the dongle. All these sets of licensing data can be integrated with each other and with the program. The goal is to create a "spider's web" that will catch any ambiguous licensing data and crash the program.

III. Do not give functions obvious names such as "licenseDecryption", "crashingMechanism", or "weHaveBeenHacked"[23]. Hackers know that most traditional programmers use top down methods, that they prefer creating multiple functions to do various tasks, and use these functions over and over again. Do not create functions for licensing tasks! Usually a hacker can just "NOP" out a licensing function (or make it return "1" or "0") without it having any effect on the program. Make sure this is not possible. Integrate the licensing code with the programming code, such as in functions which control rendering of graphics, or with functions that allocate memory for crucial processes. Most importantly, do not create simple validation functions and use these validation functions repeatedly. Use inconspicuous variable names for licensing error flags and any other variables related to the licensing mechanism.

IV. If anti - debugging mechanisms have been put in place, do not immediately

---

[22] The program may directly ask the user for this information at runtime.

[23] As trivial as this suggestion may seem, a French CAD program actually used the "weHaveBeenHacked" function name (in French). It was used (unsuccessfully) as a checking mechanism to determine whether certain license data was legitimate or had been manipulated. The availability of online language translator makes it possible for any hacker, whatever his or her origin, to translate function names.

notify the hacker that the debugger has been found lurking in memory. Set a flag somewhere. Run the program as if nothing is the matter and then crash the program randomly. This is legal if during the installation of the program, or in the "readme" documentation, the author states something similar to "This program may not operate properly if debuggers are in use". Similarly if the author decides to create a time limited protection (although I greatly advise against this) for demo or shareware versions, a warning such as "This program may not operate properly after 30 days of evaluation" can also be placed as a prerequisite to beginning the installation process. The author may then use whatever discretion when deciding how to kill the program after 30 days of use.

V. Delay the actions that are taken when improper licensing data has been found. A smart trick to play on a hacker is to immediately validate entered license data using easy to debug compares. The hacker, analyzing the debugged code, can and will only create data to fit the comparison routines. Then notify the RE that valid or invalid license data has been entered (e.g. via a simple message box). However, also incorporate the license data into the program using alternative algorithms in locations or functions that have nothing to do with licensing process. Create other algorithms to more "deeply" check the license. For example the original easy to reverse algorithm may have only checked for ten characters and checked the third character to ensure it was an integer. Elsewhere the licensing data can be verified as having a length of fifteen characters and that the third character (which must be an integer) is divisible by three. Set an inconspicuous error flag and crash the program at some later random time if everything does not fall into place. In general, make a habit of putting error checks in many locations, and putting pieces of the licensing data in many locations in memory. Remember that this entire strategy can and should be done with licensing data from any source, be it dongle, key file, or data entered by the user at runtime.

VI. Using scripted languages such as Visual Basic (compiled to pcode rather than native

code), InstallShield[24], Java, and the Microsoft Data Executable format (MDE) have both their disadvantages and advantages. Scripted languages cannot be understood simply by debugging them into their respective irreducible machine language code. This is for various reasons. Scripted languages have their own interpreters (e.g. MSVBVM60.DLL, VBA324.DLL, Java Runtime Engine or the JRE). The program sends something similar to a stack of instructions to the interpreter which decodes the stack of data and executes the program. So what the hacker spends most of the time debugging is not code within the data module but rather code in the interpreter. Thus, altering the interpreter will have no effect on the program itself, but may cause crashes and undesirable results. The hacker is forced to attempt to reverse the protection algorithms by deciphering the entire scripting language itself (by understanding how the data sent to the interpreter is actually interpreted).

The bad news is that to my knowledge, most scripted languages have their own disassemblers and debuggers. Pcode Visual Basic debuggers[25] and disassemblers have been created, as have decompilers for InstallShield and Java[26]. I have not seen a MDE disassembler, decompiler, or debugger. The MDE programming language is however quite limited and inflexible, and is mainly used for software that interacts with data sources (MDB's, etc). Scripted languages are also usually very slow (and bloated in size), since the code is not compiled into the fastest and most efficient available machine language algorithm (as in C).

7) My protection is just fine and dandy, but my program is simply being pirated!

---

[24] InstallShield is a scripting language used mainly in creating software installers.

[25] See http://vacarescu.addr.com/WkT/vbdebug/.

[26] See section entitled "Setup decompilers" and "Java" at http://linux20368.dn.net/protools/decompilers.htm.

Let it be assumed that only free, hundred percent crippled shareware and demo versions of programs are offered on the Internet. The software author does not have to worry that his program will be hacked from this aspect (unless the hacker is simply trying to "rip" code)[27]. There will however always be a customer who decides to share his purchased copy of the full program with someone else. For this reason, each software package received by the customer should be made *unique* with respect to *invisible* and *integrated* licensing mechanisms mentioned earlier in this paper (e.g. in the form of hidden key files, serial numbers, etc). In this manner the software package (the code itself) can be "watermarked" as if it were legal tender. Some CD's made today even have "holes", or large areas where nothing has been burned, wedged between regular data. Unfortunately, the advent of programs such as CloneCD and other cloning technologies have made most copy protections useless[28]. This is the problem with piracy today. People do not regard software, videos, and music as being directly equivalent to money. The ubiquity of copying and sharing has made it legal in the eyes of many.

How can the government curb the normalization of the piracy of software and other media? Simple, make it as illegal as counterfeiting money. Although the idea may be simple, the current difficulty is tracking pirated versions. A few years ago the

---

[27] This is not altogether true since it is possible for a hacker to inject code (from an uncrippled version or original sources) into the crippled version making it act as if it were the real deal, but this is usually an exercise in programming and reverse engineering. For people interested simply in the program itself, it is much easier to acquire a pirated version.

[28] In Australia the cloning of CD's has recently been publicly commercialized through the use of machines similar in concept to paper copying machines. It is believed that the end user is solely responsible for how the material is ultimately used.

common data pirate would not have thought it feasible to share or distribute “DVD Rips” or VCD’s. Advancements in data storage and Internet bandwidth technologies have made this possible for everyone (but mostly for people attending colleges and universities on Internet2 connections). In contrast, corporations and the government have been proceeding relatively slowly in the incorporation of this technology for its uses in copyright enforcement. I however, foresee that it will one day be possible to keep track of almost every legitimate (and illegitimate) copy of Microsoft Windows and Adobe Photoshop (possibly the two most prolifically pirated programs) installed on computers. This may seem far - fetched, but is possible granted how interconnected society is becoming through the Internet. Who would have thought that household picture frames could download and display images from Kodak.com?

The Internet began as a government military project and thus I believe it is possible for the government to regain control of areas that have become chaotic. Piracy, theft, and fraud occurs online almost (less than) every second, yet it is “anonymized” and thus becomes invisible or undetectable. It is however, possible to defeat the pirates and hackers at their own game with the help of skilled individuals who understand how such people think and operate.