I. J. Computer Network and Information Security, 2012, 12, 1-17
Published Online November 2012 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2012.12.01

Software Activation Using Multithreading

Jianrui Zhang and Mark Stamp
San Jose State University, San Jose, California
stamp@cs.sjsu.edu

***Abstract*—**Software activation is an anti-piracy technology designed to verify that software products have been legitimately licensed. Activation should be quick and simple while simultaneously being secure and protecting customer privacy. The most common form of software activation is for the user to enter a legitimate product serial number. However, software activation based on serial numbers appears to be weak, since cracks for many programs are readily available on the Internet. Users can employ such cracks to bypass software activation.

Serial number verification logic usually executes sequentially in a single thread. Such an approach is relatively easy to break since attackers can trace the code to understand how the logic works. In this paper, we develop a practical multi-threaded verification design. Our results show that by proper use of multi-threading, the amount of traceable code in a debugger can be reduced to a low percentage of the total and the traceable code in each run can differ as well. This makes it significantly more difficult for an attacker to reverse engineer the code as a means of bypassing a security check. Finally, we attempt to quantify the increased effort needed to break our verification logic.

***Index Terms*—**Software security; activation; piracy; reverse engineering; multithreading

I. Introduction

There are a vast number of software products available for all kinds of needs. Among these, many are distributed for free and/or as open source, while many require that users pay. Many commercial software products provide trial versions free of charge so that users can try out the product before buying—some form of activation is required to obtain the full version of the software. The trial version usually has reduced functionality and/or usage limits. However, the trial version binary usually includes all of the code for the full version.

Most software products employ a serial number for protection. Since the trial version has the same binary code as the full version, it is possible to crack the trial version and remove the limitations to obtain the full version. In fact, software products are often cracked by hackers who modify, or patch, the activation mechanism. After breaking the activation mechanism, a motivated hacker can create a key generator (or, simply, KeyGen) or patches to distribute via the Internet so that other users can easily obtain the full version of the code without paying.

KeyGens or patches for many popular software products are readily available [20,21]. As a result, in many countries, software piracy is rampant. For example, it is thought that a majority of computers in China run pirated versions of Microsoft Windows. In fact, it was reported that Windows 7 was cracked several months *before* its official release [2]. Figure 1 shows the estimated level of software piracy in various countries.

Our research focuses on developing an improved serial number checking mechanism. The goal is to make the hacker's task more difficult. This paper is organized as follows. Section 2 discusses several common techniques employed in software activation while Section 3 focuses on serial numbers as an activation mechanism. In Section 4 we discuss anti-reverse engineering techniques, and in Section 5 we provide details on our software activation design. Section 6 covers our testing setup and results. Finally, Section 7 provides a conclusion and suggestions for future work.

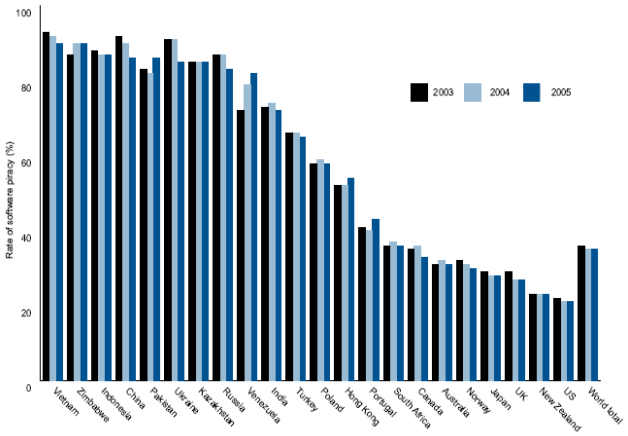

Figure 1. Level of Software Piracy [22]

II. Software Activation

Software activation is used primarily as a way to make users pay for the software they use; this is how software companies make money to continue their business. Today, consumers can try various software packages before they decide which to buy. Software

vendors attempt to make the trial version attractive enough to entice consumers while setting some significant limitation so that users will eventually feel compelled to pay for the software. A strong software activation mechanism would reduce piracy and thereby help developers to get paid for the use of their software.

Next, we cover various kinds of protection that are commonly used as part of software activation. Then we briefly discuss different software activation methods.

2.1 Software protection mechanisms

Perhaps the simplest "protection" is a nag screen that pops up each time the software is started and, for example, reminds the user of the number of days that the software has been used without paying, and provides registration information. Such an approach relies on the slight annoyance created by the nag screen as well as playing on the user's conscience. Apparently, many users are not bothered by their conscience, since it is not uncommon to find cracks that simply remove nag screens.

At the opposite extreme from a nag screen, the trial version is sometimes a completely different program than the full version. That is, the full functionality is not available in the trial version, so there is no point in directly hacking it. While this is the strongest possible method of software "activation", it does require the developer to maintain two distinct copies of the code base. In addition, this approach requires a second download when the code is purchased, which might annoy some users. This approach appears to be reserved primarily for relatively expensive software.

Modern shareware often limits the number of days a user can access the trial version of the product. The goal is to make the software unusable after the time limit has reached. If a user likes the particular software and uses it for an important purpose, the user could purchase the software and continue to use it. According to [3], this type of protection is also fairly easy to break. For example, CD Key Generator from Jedisware [23], utilizes this kind of protection in its trial version and limits usage to only five days. However, a moderately skilled attacker could crack CD Key Generator in a few hours [4].

In the commercial realm, usage and time limits are not common. Instead, expiration dates are more the usual practice. In addition, an expiration date is often used on beta products (such as the various beta versions of Microsoft Windows) in an effort to coerce users to buy the full version once it is released. One simple (and surprisingly effective) way to break this kind of protection is to reset the system clock to a time before the expiration date.

It is common practice to provide users with a trial version with reduced functionality, which is sometimes referred to as "crippleware". For example, the trial version of Cyberlink's PowerDVD [24] lets users play back DVD movies for five minutes or less, while the full version has no such restriction. In most cases, the executable for the trial version is the same as that for the full version, which makes it possible to break the protection and turn the trial version into a fully functional version. In fact, it is common practice for hackers to break such protection and distribute the cracked versions on the Internet.

There are software products that use the presence of a disk (containing some critical information) in the CD-ROM drive to start the program. This method is mostly used by the computer game industry and, in general, is considered easy to crack [3]. As evidence of this, it is possible to find cracks online for virtually all popular game titles.

Encryption and hashing have potential roles to play in software activation. An attacker cannot read encrypted code, so encryption can foil disassembly. Of course, the code must be decrypted before it can execute, which makes it possible for an attacker to obtain the decrypted code, but some additional work may be required. On the other hand, hashing can be used as an integrity check—the hashing does not obscure the code, but instead it is used to detect modifications and thereby make patching more difficult. When these cryptographic techniques are used, the protection is usually applied only to security-critical parts of the logic because these are the hot spots for potential attacks.

There are many software products that employ more than one of the techniques mentioned above. Different protection methods used in combination can reinforce each other and make cracking significantly more difficult. We have more to say about this when we discuss our design in Section 5, below.

2.2 Software activation mechanisms

Serial numbers are the most popular method for activating software. That is, the user types in a serial number obtained from the vendor, in effect, purchasing a legitimate copy of the software. In some cases, a username is also needed.

There are two common ways to distribute serial numbers. The first option is to distribute the serial number along with the media containing the installation package. A second option is via email—after purchasing the product (usually online), the vendor sends an email confirmation to the user along with a serial number for the product. Email distribution is commonly used for

shareware. In the next section, we discuss serial numbers in more detail.

An activation file is sometimes used, although this approach is not common. This method usually works in conjunction with software distribution via a download. A consumer purchases the software online at the vender's website, and the vendor sends an email to the user with an activation file attached. After receiving the activation file, typically, the user must save the activation file to some specified location. When the software launches, it checks for the existence of a valid activation file—if the file is found, the software installs as a full version. Activation files may contain information that is unique to each user. For example, RarLab's popular WinRAR [26] uses an activation file.

Activation by hardware key is sometimes used, but it is one of the least common methods in use today. This approach requires the presence of some special hardware device before the software will function [5]. This kind of activation can be difficult to break since it is not easy to determine what the hardware key does. For example, code on the hardware device may be necessary for some crucial calculation performed by the software. Without access to the code on the hardware key, an attacker would have to fill in gaps in the available code, which would generally be a futile task. Even with access to the hardware key, stitching together the pieces to create a stand-alone functioning piece of code could be challenging.

A hardware key could be a USB key or, ideally, a smart card. For example, the Bank of China requires a USB drive to activate its online banking software [7]. The advantages of using a smart card include readily available cryptography and tamper resistant hardware—any communication with the smart card is cryptographically secured and the smart card is able to lock or destroy the data it contains if authentication repeatedly fails [6].

Pre-activation by the vendor is employed when software products are bundled with a new computer. For example, Microsoft's Windows operating system is the most widely pre-activated software.

For Microsoft Windows, activation information is stored in the BIOS on the motherboard and the OS checks the BIOS for the presence of this information. Microsoft Windows is a popular target for attack, and hackers often exploit this activation method.

Table 1 gives a comparison of various software activation methods, including the pros and cons of each. This information is essentially a summary of the "lessons" in [3].

Table 1. Methods for Software Activation

Method	Popularity	Convenience	Strength
Serial Number	Very popular	Convenient	Relatively weak
Activation File	Used, but not common	Somewhat convenient	Relatively weak
Hardware Key	Not common today	Not convenient	Relatively effective
Pre-activation at vendor	Popular for OSs	Convenient	Relatively effective

III. Serial Numbers

Serial numbers are the most popular method of activating software products. Serial numbers, which are alphanumeric strings, are sometimes known as CD keys, product keys or activation codes. Ideally, each legal copy a software product should be activated by a unique serial number, although this is often not the case, particularly for shareware.

3.1 Checking serial numbers

Most software products only check the serial number once, when it is initially entered. In this approach, after a serial number is deemed valid, it will, in effect, be valid forever. Dual checking is an attempt to improve on the one-time checking mechanism. This method is used in Adobe products, such as Photoshop, which requires users to go online (or contact Adobe by phone) to obtain a second activation code. The second activation code is necessary to complete the activation process; see Figure 2.

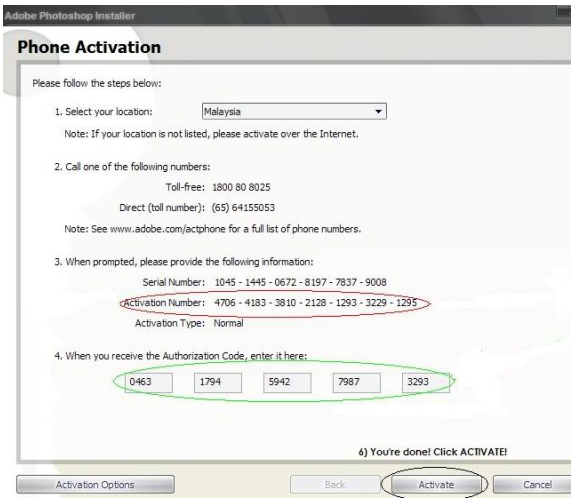

Figure 2. Adobe Photoshop's 2-layer Activation [25]

By requiring users to contact the vendor, the vendor is able to validate whether the first serial number is valid. The vendor thus has a better chance of being able to keep detect serial number fraud. Sometimes serial numbers are checked repeatedly over time. Microsoft employs this method in its Windows XP and later operating systems. When users download critical updates, Microsoft will check whether the current OS is a legal copy by using its GenuineAdvantage software. Figure 3 shows Microsoft's GenuineAdvantage in action.

Figure 3. Online Software Validation

The advantage of repeated online software validation is that it provides a vendor with multiple chances to detect piracy. The downside of this approach is that a vendor must entice users to repeatedly "check in" with the vendor. For operating systems, this is plausible (updates, patches, etc.), but for most software products, users would have little incentive to do so.

3.2 Entering serial numbers

A common way to enter a serial number is during installation. This method is usually used by software without trial versions. Cracking such software may be more difficult (since the attacker lacks context), but it is doable.

Many software products allow users to enter a serial number after installing the software. This makes life somewhat easier for hackers, since they can usually zero in on the important parts of the code.

Some software products have serial numbers built into hardware keys. In this case, users do not see the serial number at all. As discussed above, this makes it difficult to break the software activation mechanism. However, this method is not widely used today.

3.3 Generating serial numbers

There are many ways to generate and store serial numbers. How this is done directly affects how easy or difficult it is to break serial number checking mechanism.

If software developers do not have much experience in this field, they may be better off using third party products for protection. One company that provides such service is LogicProtect; it claims to provide "clever software activation, anti-piracy functionality and copy protection for your software" [8]. LogicProject's service description says its service is able to provide both activation and online verification [8]. This will make the overall process more robust. In essence, LogicProtect provides its service by letting developers integrate LogicProtect's DLL into their software. In its newest release (version 7.0), it even includes web service APIs, which make the online verification easier to implement.

In many cases, software companies prefer to develop their own secret algorithm for generating and checking serial numbers. The idea behind this practice is that the "secret algorithm" is supposed to be difficult to break because no one from the outside knows about it; however, this idea contradicts Kirchhoff's principle [9]. In fact, the majority of serial number generation and checking algorithms are broken by hackers. Once the part of the code responsible for serial number generation is identified, hackers can simply "rip" out such code and use it to create a KeyGen for that software product [10]. Among different secret algorithms, use of hash functions is one of the favorites.

Sometimes software developers use third party software products to generate serial numbers and develop their own code to verify the serial numbers. Jedisware CD Key Generator is one software product that can generate serial numbers of various lengths and formats (such as use of hyphens, numbers only, and so on). The full version of CD Key Generator allows users to save the generated serial numbers in a file or in a few data structures such as array or arraylist. Ironically, CD Key Generator itself is not good at serial number checking—it stores all 5000 valid serial numbers as an array of strings in the software and simply compares against all stored valid serial numbers to check for validity [4]. Clearly, it is a bad idea to store valid serial numbers in source code, since these will be obvious to anyone who reverse engineers the exe.

3.4 KeyGen

A key generator, or KeyGen, is a hacker-developed tool that is used to generate valid serial numbers for a specific piece of software. Such a tool enables any user—regardless of skill level—to create a valid serial number, which can then be used to illegally activate the software. A Google search is often all that is needed to find a KeyGen for a particular software product.

There are two common ways to create a KeyGen:

1. Analyze and recreate the underlying algorithm by studying the program disassembly.
2. "Rip" the assembly code from the disassembly and use it directly.

Both of these methods require identifying the section of code responsible for checking the serial number, but the first method is far more labor intensive, since the attacker must study of the code and reconstruct the algorithm. In contrast, the second method only requires a copy and paste of the disassembly and, generally, some minor fixes to get the code to work. In some cases, additional checks may be required (either in the code or by contacting the vendor) which are independent of the KeyGen.

IV. Anti-reversing Techniques

In this section, we discuss various anti-reversing techniques, including anti-tampering, anti-debugging, and code obfuscation. We single out multithreading for more discussion, since it will figure prominently in the remainder of this paper.

4.1 Anti-tampering techniques

Developers can also employ techniques to make their code more difficult to modify. Such anti-tampering techniques can be used with or without code obfuscation. Hashing a binary executable is one way to detect code patching. One problem with this is that the hash value must be available to do the check, which makes it subject to attack.

Ideally, hashing should be applied to code after it is loaded into memory. Such an approach could effectively prevent hackers from using debuggers to modify code at runtime in order to change execution flow. However, hashing the executable after it has loaded is difficult to implement in practice, particularly on machines that employ address space layout randomization (ASLR).

4.2 Detecting a debugger

Hackers must use debuggers to successfully understand the design of an activation mechanism and to determine how to patch the code. Therefore, if we can make debugging more difficult, we can make the attacker's job more difficult.

IsDebuggerPresent() is a system function in the Microsoft development library. If a process is started by a debugger, calling this function can detect the presence of the debugger. However, if a debugger is attached to a process after it is started, calls to this function return false.

The IsDebuggerPresent() function can be easily identified by modern debuggers, as illustrated in Figure 4 using OllyDbg. Hackers can easily disable calls to this function and bypass the check, as shown in Figure 5 and, consequently, this method is not particularly effective.

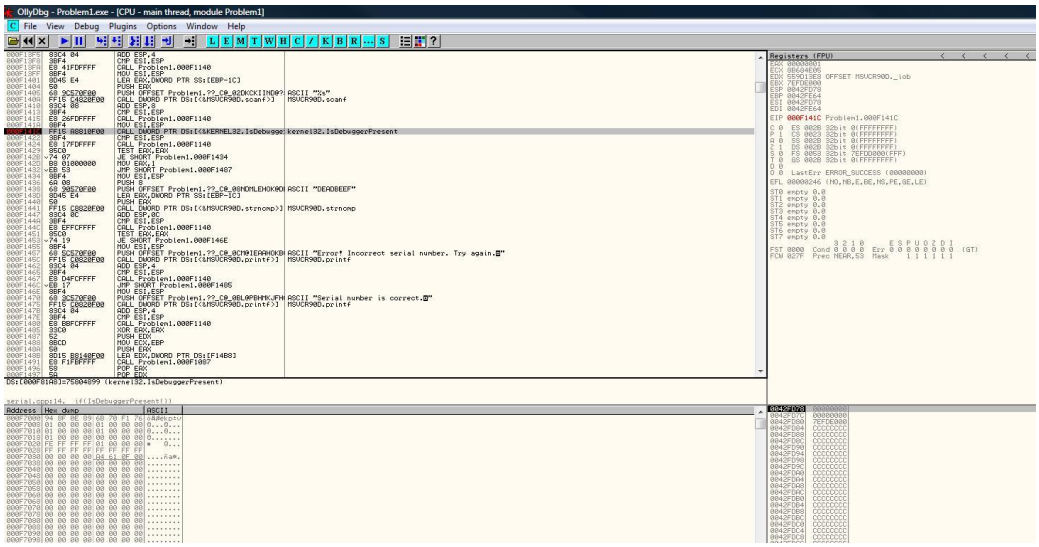

Figure 4. Identifying IsDebuggerPresent()

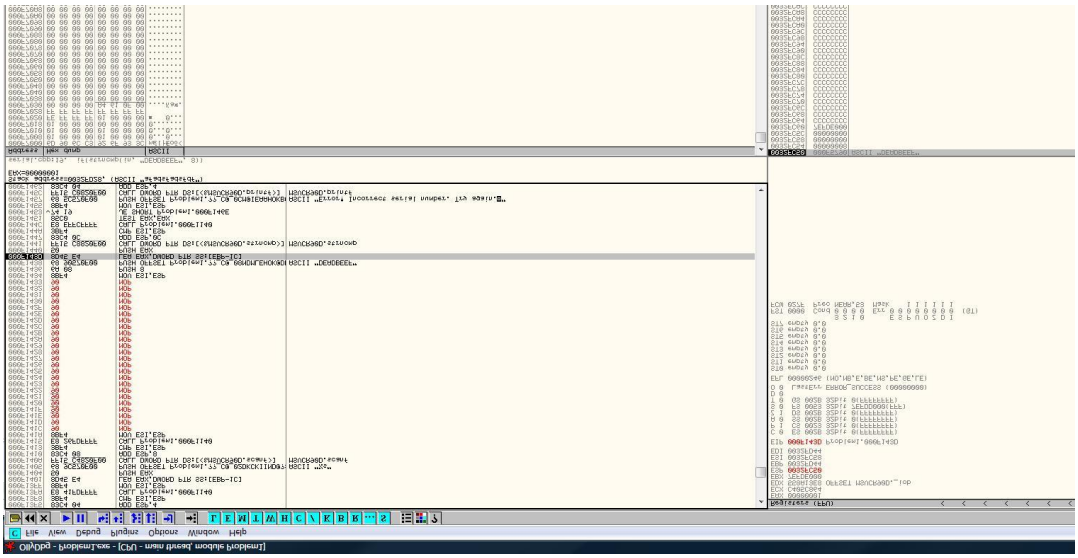

Figure 5. Bypassing IsDebuggerPresent()

Developers can write their own code to detect a debugger at runtime. One such method is to check the run time of a segment of code—if a debugger is used, the run time will likely be much longer than if not. Developers can use trial and error to determine the normal run time of a block of code.

4.3 Code obfuscation

Developers can add in various well-designed assembly codes to confuse disassemblers. However, our research found that modern disassemblers are smart enough to deal with this tactic. At best, only a few lines of disassembled code can be confused, hence proving this method of less value. In Figure 6, the boxed line of code in red shows the only line of assembly code that got messed up.

Insertion of junk code into meaningful code is intended to confuse hackers. Junk code works by causing hackers to spend more time studying useless code as well as divert their attention from good code. Our research found that when much junk code had been inserted, it may not be possible to identify the good code from the bad. It definitely took significantly much more time in hacking efforts. Overall, this technique can be very effective. In this section, we will discuss 3 kinds of junk code: junk logic, metamorphic code, and recursion.

Junk logic is junk code added in the code section. Common examples include adding useless instructions and mixing them together with useful code. This provides protection at the expense of run time. Depending on how much junk code is inserted, run time overhead can be significant.

Junk data refers to useless variables in source code. Its purpose and use is more or less like junk logic, except it may not have considerable overhead in run time.

Metamorphic code is another possible protection technique. A metamorphic engine mutates code while maintaining the original function [18]. While this technique was invented by virus writers, it can be applied as a means to protect code by making reverse engineering more challenging.

Recursion is another useful obfuscation technique. Recursive function calls are good for significantly increasing the stack size because many parameters and return addresses will be placed onto the stack in the process. This can effectively disrupt a hacker's view of information stored on the stack. One downside with this technique is recursive functions are usually short in length of code and hence can be easily spotted and understood. If the recursion does not do anything useful, hackers can simply disable them.

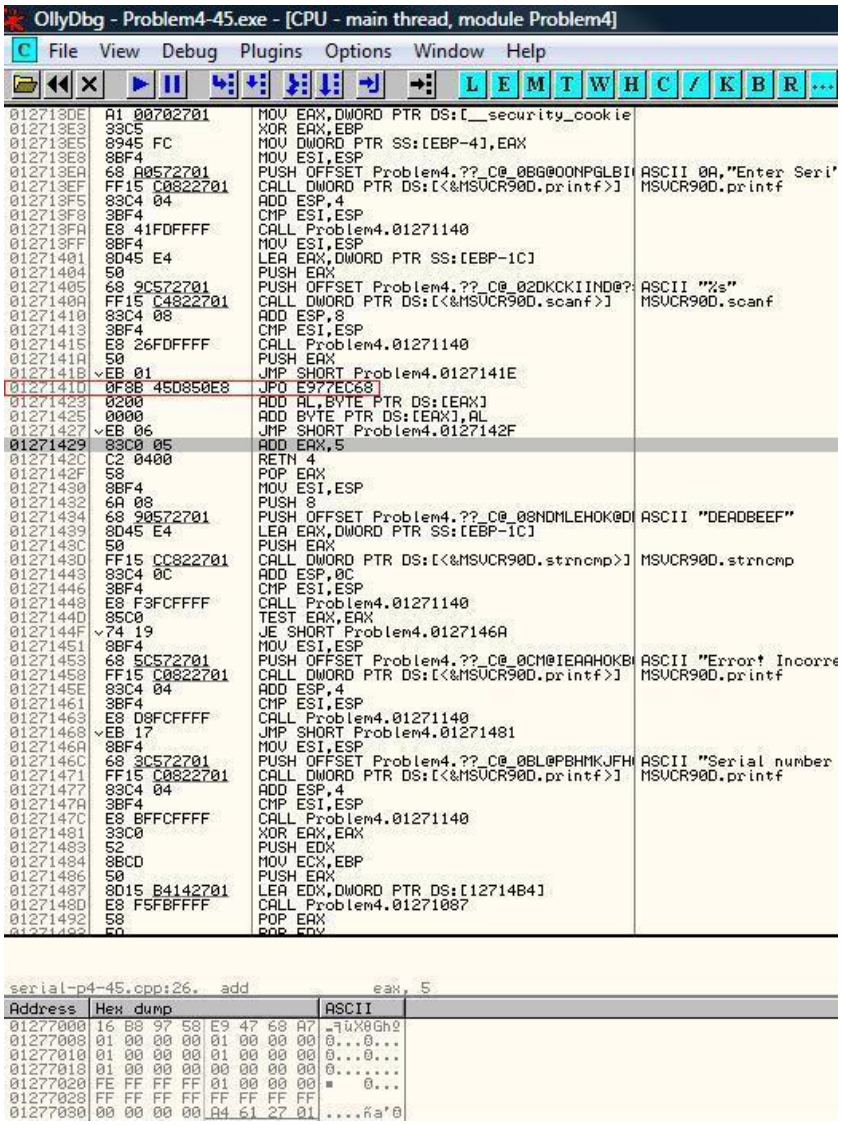

Figure 6. Confusing a Disassembler

String obfuscation can be used to hide certain types of important information. Simple encryption techniques, such as XOR or one time padding, can accomplish this purpose. One problem with simple encryption is that a hacker can get information out of the cipher text based on its length. To make string obfuscation more effective, developers should to use a different length for the encrypted strings compared to the original ones. Another problem with this technique is that hackers are not usually interested in the strings themselves; rather, they want to know how and where the strings are used. Checking mechanisms often display messages to users after they input serial numbers to indicate success or failure; these messages often give out the location of checking mechanism. Given that hackers are more interested in identifying locations of checking mechanism, they can trace system function calls related to outputting messages, such as "print" or "Messagebox.show()" instead of focusing on trying to work out the obfuscation method. In this regard, string obfuscation may not provide much benefit for our purpose. Figure 7 gives an example of a debugger identifying system function called "fopen" and using it to find out string "readme.txt" as file name from EAX register.

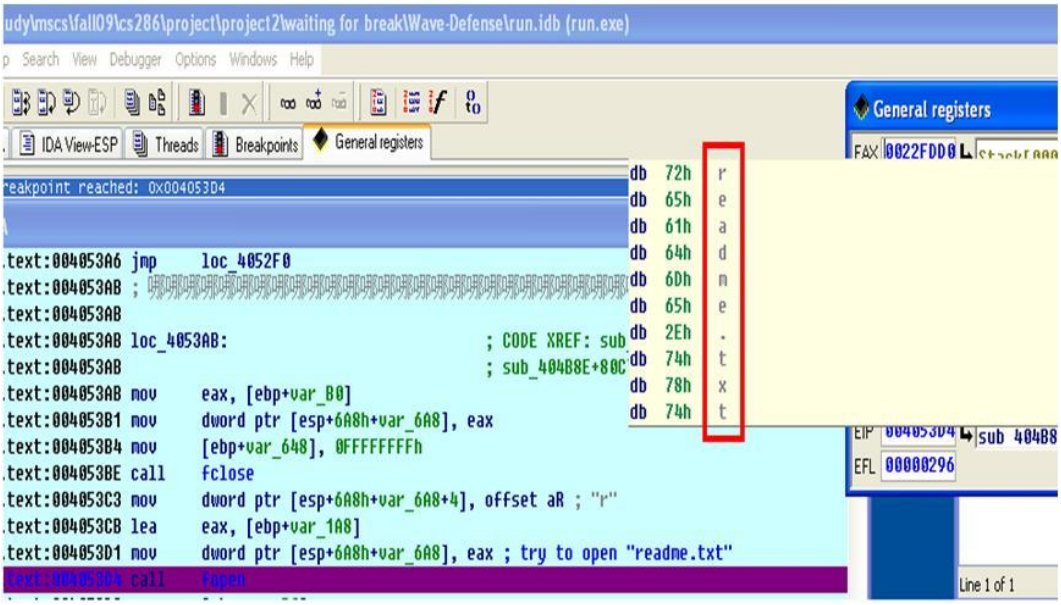

Figure 7. Obfuscated String in Clear Text

An opaque predicate is a comparison whose outcome is either always true or always false and known to the developer at development but not program at run time. Using opaque predicates increases the number of branches of code hackers need to trace, which can be very time consuming. Sometimes opaque predicates may actually be useless as they can be easy to spot; for example, if opaque predicates make use of floating point calculation in an algorithm that only uses integer calculation (or non-floating point calculation in general, as often is the case for serial number checking), hackers would know what code to skip. In contrast, using opaque predicates in places where they should not be found may lead to a revelation of important logic. After tracing code a few times, hackers can realize their existence base on execution flow as well.

Control flow obfuscation refers to code executing in strange order or, at least, appears as a strange order. This is usually accomplished by using many "jumps." In essence, this is used to break locality of code. Psychologically, people would think code blocks next or close to each other are related and often are executed sequentially. Once locality is broken, hackers can feel lost when they have to jump through different places in order to trace code. Figure 8 shows how complex control flow can be by adding a considerable amount of junk code into one subroutine.

Windows events are directly related to graphical user interface, commonly known as GUI. Here we use windows events to obfuscate the execution flow, more or less like using multithreading. Windows events are raised by users through interaction with a GUI and processed by an interface thread (sometimes known as an event thread). Developers can take advantage of this by handling multiple events in the code so that execution will jump from one place to another sporadically making hackers feel lost. Events, such as mouse movements, will be triggered many times, which can certainly annoy hackers.

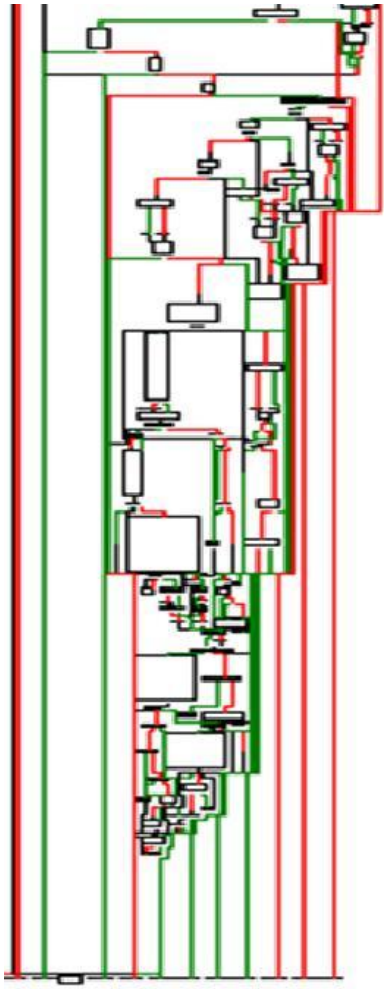

Figure 8. Subroutine Flowchart

4.4 Multithreading

The original purpose of having a multithreaded application is to parallelize some of the logic and have the threads execute concurrently to increase overall efficiency. Here we use multithreading to increase difficulty of debugging.

It is inherently difficult to debug a multithreaded program even if its developers have the source code due to a variety of reasons, such as data synchronization and so on. The difficulty arises from the fact that only an operating system has control over when and which thread runs, but not the application itself and hence not the developers either. In addition, debug mode and release mode may yield different results for the same piece of code. For example, if the developers did not initially synchronize data correctly, the release mode may yield incorrect results whereas nothing may seem wrong in debug mode because the debug mode may force synchronization as it has to display the result to the viewer.

For our purpose, we can use multiple threads to do the work concurrently so that hackers cannot easily single step through code to find out how the logic works, since validation may have been completed elsewhere. Table 2 gives our view on the relative effectiveness of various anti-reversing techniques.

Table 2. Comparison of Effectiveness of Different Anti-Reversing Techniques

Method	Relative Effectiveness	Pro	Con
Junk code	Strong	Makes code hard to trace	Performance
Recursion	Weak	Makes stack large	Performance
Hashing	Moderate	Can detect changes to code	Performance
String obfuscation	Weak	Hard to find critical logic	Easily detected
Opaque predicate	Weak	More branches to follow	Performance
Control flow obfuscation	Strong	Breaks up proximity	Code complexity
Multiple validation logic	Moderate	Reduce single point failure	Code complexity
Multi-threading	Strong	Very difficult to trace code	Major code complexity
Window events	Moderate	Hide sequential execution	Performance

V. Proposed Design

This section will propose a new design, along with testing results of the new design. Then we discuss the techniques used in our proposed new design.

5.1 Design Considerations

In this section, we will outline several techniques considered but excluded from the new design. One way to use hardware keys is to use the hardware device to perform part of the computation; similarly, we can do part of computation online, such as using web services. In this approach, the installed local copy does not have full functionality. The server side can check for proper licensing before completing requested computation. This way, activation mechanism is nearly hack-proof because hackers can't trace (step through) the checking logic located on server side; however, such activation mechanism is way too complicated to implement, not to mention significant overhead and slowness, which renders this method impractical in most applications.

Encrypting executable is a strong anti-disassembling method. But this is extremely difficult to implement in practice, especially with new security features built into current operating systems (OS). Storing an encryption key safely is another issue.

It is nearly impossible to do reverse engineering work without a debugger, so disabling them (in one way or another) seems to be an attractive choice. But in practice, it is very difficult, if possible at all, to disable use of a debugger. The core issue here revolves around the inability to determine presence of a debugger effectively and accurately, partially due to new hardware architecture and new OS security features.

5.2 Design

Instead of requiring a user type in a program serial number from a keyboard, a license file will be used. The license file should be generated by the software vendor,

and distributed to users via email; users should then save the license file in a proper place on their hard drives.

The license file should be encrypted using a strong encryption algorithm, such as Advanced Encryption Standard (AES), with an/a encryption/decryption key derived from a password, one that is only known to the vendor and user (each user will decide their own password during registration process). In this design, the format of the license file is XML, and contains information such as username, the hash value of program's binary, a serial number, and necessary validation information. Other information, such as trial expiration date, can be also included if necessary.

The hash value of the program's binary is intended to deter modification of the program by attackers. A Hashed Message Authentication Code (HMAC) algorithm is used to calculate the hash value with a key derived from the user's password.

The reason for using a license file, instead of manual user input, is to make it more difficult to locate the corresponding code responsible for validation. With breakpoints smartly set in a debugger, an attacker may be able to quickly find out roughly the beginning and end of a code region of interest, and then concentrate on that particular area. This is possible if the debugger is able to jump to that section of the code in question when it executes. In contrast, it is difficult to discover when the code of interest executes if it does not require user interaction; additionally, hackers would have to trace code from the very beginning to find out where code of interest is located.

Using multiple threads to do work for serial number checking is the core idea in this design. The entire serial number verification process is divided into many small pieces (functions), and each piece will be run using a separate thread. Any dependency among threads can be resolved by "WaitHandle." On a high level design, the verification can be divided into 4 parts: verifying the program binary's hash value, and 3 verification logics for checking the serial number. Each of these 4 logic blocks is further divided.

There are a few reasons why multithreaded processing is chosen here over a single threaded version. First, it breaks the sequential execution flow. Even if code is broken into many pieces, the execution flow is not changed (disrupted); a hacker can still easily trace the execution to understand in which order the code is run. Once the order is known, code can be analyzed more effectively. In essence, breaking-up code and running it in a sequential order, at most, makes code tracing a bit annoying, having to jump from one place to another. Having many jumps can break attacker's sense of locality, but with analytic tools, code can be easily understood by drawing a flow chart. In contrast, using multiple threads running concurrently will fundamentally change the execution order, which makes code much more difficult to trace.

Second, multithreading is very debugger-unfriendly. Even with source code, a multi-threaded application can be very difficult to debug [19]. Timing is absolutely one of the most important factors when debugging a multi-threaded application. A bug observed in normal run may not be reproducible in debug run simply because the timing is different. Also, a debugger is not able to trace two threads at the same time, in the sense that one cannot single step through more than one section of disassembly at the same time, even if the debugger is aware of existence of other threads.

Third, it is out of anyone's control when and which thread runs; this is only determined by the operating system's (OS) task scheduler. Because of this, different runs of the same code on the same debugger may yield different execution sequence, depending on which thread the debugger is able to gain control over.

Using multiple validation logic has an obvious advantage because it may prevent a single point failure. Our design employs 4 validation logics with 2 of them being able to correct each other if an inconsistent result is detected. While this method is not foolproof, it certainly should work against attackers, as attackers will have to spend much more time locating existence of these logics and then breaking them. At the beginning of program, only 2 of the 4 logics are executed, and the other 2 are delayed according to our new design. This way, attackers may not discover the other logics even if they follow the execution flow from the start.

In our design, certain GUIs are disabled by default, and their corresponding event handlers are not registered with the event. This is used to prevent unauthorized use of some special functions, such as full functions not found in trial versions. GUIs are properly enabled and event handlers properly registered if, and only if, all validation logics determine the program is a legitimate full version (not a hacked version). They are routinely turned off and on again to prevent an attacker from enabling them at a program's start by modifying the binary code.

OnIdle is an event issued by the OS when a program is in an idle state; it allows for idle time processing of low priority tasks. When a program needs user interaction, this event will be issued very frequently, as the user is very slow compared to the hardware. When the program does not have a user focus (not being the topmost application), this event may not be issued since this entire program may not receive any CPU time. This new design utilizes the abovementioned feature of OnIdle as an anti-debugger technique and will use this event to process certain important tasks, such as synchronizing encryption/decryption keys and serial number checking.

Serial number checking takes advantage of an idle event being run very frequently, whereas key synchronization takes advantage of an idle event and can only run when a program has user focus. In the latter case, crypto keys may not be synchronized if the OnIdle function does not run, such as when the debugger windows are on top of the program's window.

With certain functions that require paying for a full version license, their results will be encrypted and then decrypted with key pairs. One key is calculated in advance at license issue time and stored in the license, while the other is derived from a serial number checking

process. If everything goes right, these two keys are identical; therefore, encrypting the result then decrypting it should not change the result. If keys do not match, the correct result will be altered in the decryption process, yielding an incorrect final result for output.

This method adds protection against unauthorized use of a full version feature when not properly licensed, but it may carry significant overhead due to crypto-operations.

Since we believe multi-threaded checking is more effective than a single threaded version in terms of anti-reversing in theory, this design will run extra threads to complicate the situation more. And these extra threads will be used in combination with deadlocks.

Deadlock refers to a situation in which 2 or more threads each holding some resources while waiting to acquire more which are held by other threads; because no thread is able to obtain all required resources to proceed, all of them will sit idle and blocked. A classic example of deadlock caused by cycle is illustrated in Figure 11.

Deadlock can work well against stepping through code in a debugger. When stepping through instructions in a debugger, one cannot move to the next instruction until the current one finishes. For example, if one tries to step over a function call that takes a long time to finish, the instruction right after the function call cannot be executed until the call returns. In this case, execution is temporarily blocked. If that function never returns, such as running an infinite loop, then the next instruction will be blocked indefinitely. In this new design, we will purposely create a deadlock situation with extra junk threads (threads that do not execute any useful work). When a debugger picks such a thread for a user to step through, it is expected that the progress will be blocked indefinitely. This technique attempts to divert an attacker from stepping through those threads that do work of real interest.

In our design, certain operations are delayed to hide its relationship with other operations. For example, one important use of this is exiting the program when checking fails to pass. Certain system calls can be easily identified by debuggers by tracing these backwards sequentially, an attacker may discover where checking is performed. By delaying a certain execution and running it in another thread, we can effectively break an attacker's sense of code locality, making backwards tracing pointless. Using this technique, we can shift comparisons away from checking logic, forcing an attacker to trace more code.

Obfuscated code is more difficult to understand, because one has to distinguish between the useful and useless code. This is often accomplished by inserting junk code and shifting code blocks around it. In this project, we hope to apply this technique to scramble code, but it is not easy to find a good polymorphic engine to accomplish this task. Xenocode's PostBuild [17] has built-in code obfuscator; we will use it without analysis of its effectiveness.

Figure 12 below shows the flow and dependency of blocks responsible for verifying the integrity of a program's binary by hashing. If modification is detected, the program will terminate itself.

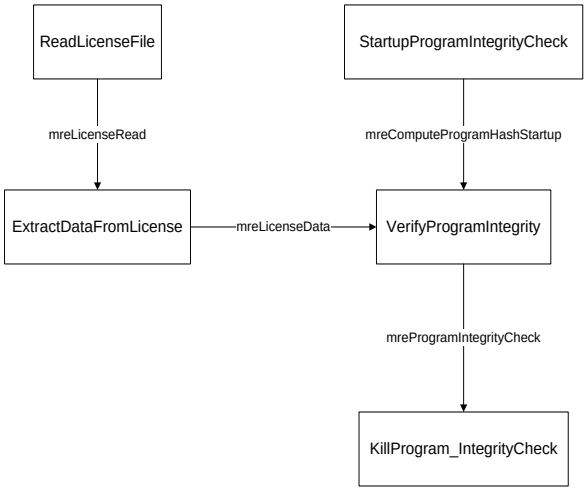

Figure 9. Block Diagram for Integrity Check

Figure 13 shows the flow and dependency of blocks responsible for verifying the serial number at program startup. If verification is passed, GUI and corresponding handlers are enabled.

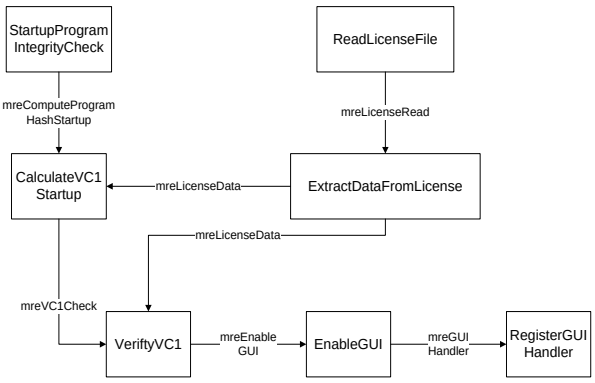

Figure 10. First Module

Figure 14 shows the flow and dependency of using a secondary module to verify a checking result obtained at program startup. If secondary checking demonstrates a different result than the startup checking, overall verification is deemed failed. In this case, the program will terminate.

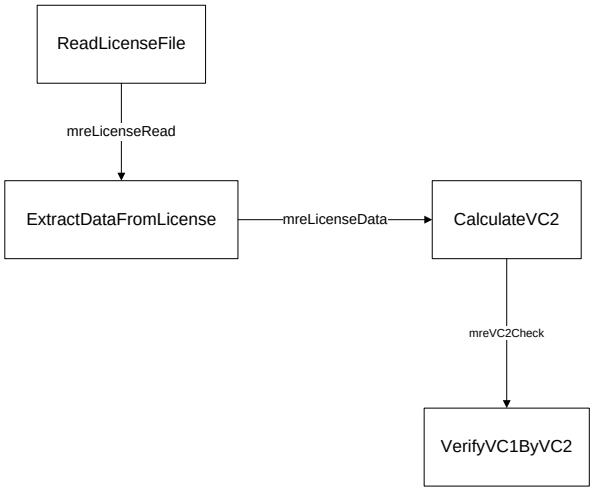

Figure 11. Second Module Verifies First Module

Figure 15 shows a flowchart of utilizing a timer to activate the 3rd verification module, whose result will be compared to that of the secondary module. If a difference is detected, overall verification is deemed failed, GUI will be disabled and handlers will be deregistered, and the program will terminate.

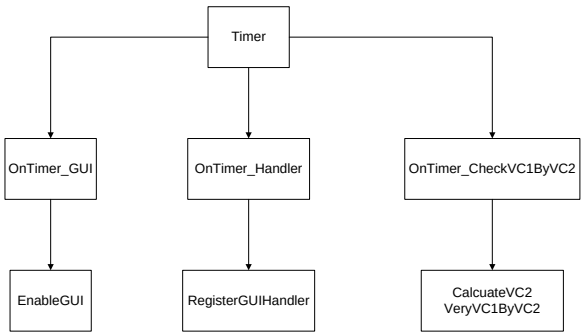

Figure 12. Third Module Checks Second Module

VI. Testing and Results

6.1 Testing Setup and Metric

A demo program was written in C#, then converted to native x86 binary using XenoCode's PostBuild, without any obfuscation applied. Microsoft Visual Studio (MSVS)'s built-in debugger will be used alongside a source code to set expectations; this would not be the real world scenario. Tests were repeated using OllyDbg and IDA Pro. These tests was the main testing. A program can be set to run in a specific mode (single threaded versus multi-threaded), and a number of junk threads can be specified.

Tests were be divided into 3 parts. Part one was the correctness of implementation. Tests in part one included testing for correct thread count, as well as the correct behavior of some functions. Part two was on comparing a single threaded version against a multi-threaded version. Testing in part two determined whether using multiple threads for checking has advantages over a single threaded version. Part three examined whether junk threads will make attacking more difficult.

In our testing, we used number of lines of disassembly that can be stepped through as the main metric. In a single threaded version, one should be able to step through all relevant code in order to analyze it, whereas in a multi-threaded version we expect only some of the code can be traced. If an attacker cannot trace and analyze all the relevant code, there is little chance the attacker can successfully break the software security.

Also, extra effort needed to implement the multi-threaded version will be considered and compared to the single threaded version.

Finally, XenoCode's obfuscator was simply evaluated, by comparing how much of the disassembled code are different.

6.2 Test Results

In testing our implementation, we paid particular attention to the correctness of threads. Table 3 below summarizes results of different test runs as they relate to thread issues.

Table 3. Demo Program's Thread Count in Various Running Modes

Thread Mode	Number of Junk Threads	Observation
Single threaded	N/A	Program runs on 9 threads minimum (GC + GUI + Timers + asynchronous event firing, and so on). Max was 12 as reported by WTM.
Multi threaded	0	WTM reported a max of 12 threads running at the same time. Thread count gradually falls to 9 according Windows Task Manager (WTM), which is the similar to single threaded mode. This makes sense too since when checking is done, most extra threads are terminated. Theoretically, the program should launch 10 individual threads, but it appears that they do not all run at the same time.
Multithreaded	5	WTM reported a max of 17 threads running at the same time; it falls to 14 after a while. Total count is 17 because of 5 junk threads.
Multi threaded	10	WTM reported a max of 22 threads running at the same time; it falls to 19 after a while. Total count is 22 because of 10 junk threads.
Multithreaded	15	WTM reported a max of 27 threads running at the same time; it falls to 24 after a while. Total count is 27 because of 15 junk threads.

The thread counts in the Table 3 are consistent, assuming 9 threads are needed to run the application on average after checking is completed. Running code in debuggers has the same count as running it without debuggers; therefore, implementation of threading is correct.

Our demo software was tested with Microsoft Visual Studio. The tests shown in Table 4 are done with MSVS's debugger with source code. The reason for using this testing environment is so that we can set breakpoints correctly and track which function is being executed. In other words, this is for the purpose of convenience and to set our expectation when debugging in other environments; without such convenience, debugging can only be substantially more difficult (this should be the

best testing scenario possible). Table 4 summarizes testing results in various scenarios.

Table 4. Testing Scenarios Using MSVS Debugger

Test Case Number	Observation
1	Single Threaded, no junk thread, break on all relevant functions. Unable to proceed to other functions because Idle function runs continuously and this is the function captured debugger's attention all the time. GUI is launched, but unable to interact with it because Idle is constantly running.
2	Single Threaded, no junk thread, break on all relevant functions except Idle. Without Idle interfering, GUI is launched, and can be interacted with normally. Checking is done sequentially in the right order as specified. All parts of code can be traced. A frequent timer event can severely disrupt debugger process, as relevant functions run all the time. All handlers of timer event can be debugged, as long as breakpoints are set for them. Present of timer did not affect debugging code relevant checking functions, because they do not take effect until initial checking is done.
3	Single Threaded, no junk thread, break on all relevant function, but Idle added in later. As long as the first breakpoint for checking is reached before setting the breakpoint of Idle, checking can be traced as in case 2 above, but timer events cannot be traced due to constant running of Idle.
4	Multi Threaded, 0 junk threads, break on all relevant functions. Observation is identical to case 1 above, which matches expectation
5	Multi Threaded, 0 junk threads, break on all relevant functions except Idle. The first function can be partially traced, a few other functions can be traced partially at random (non-deterministic about how much of a function can be traced). Timer events can interfere with normal tracing
6	Multi Threaded, 0 junk threads, break on all relevant function, but Idle added in later. Result similar to case 5, except Idle will disrupt code execution more severely compared to case 5 above. In essence, checking is executed interleavingly with Idle.
7	Multi Threaded, 2 junk threads (minimum needed to create deadlock), break on all relevant functions. Deadlock situation is successfully created. After deadlock, only Idle can be traced, no checking can be traced.
8	Multi Threaded, 2 junk threads, break on all relevant functions except Idle. Deadlock situation created, but checking can be traced like in case 5 above.
9	Multi Threaded, 2 junk threads, break on all relevant function, but Idle added in later. If added too soon, then like case 7 above; if added late enough, then like case 8 above.

Additional tests were performed with more junk threads (with numbers being 5, 10, and 15) in the same setup as test case 7, 8, and 9 in Table 4, and the same results were obtained correspondingly. In this case, more threads being deadlocked added no extra benefits. In fact, the presence of a deadlock added no more difficulty compared to just being multithreaded in this particular testing environment. This is due to the fact that Microsoft's debugger (with source code) can smartly execute code in an interleaving manner, allowing the execution to change from one thread to another, although it is out of the user's control which thread is executed and when.

Single breakpoint was also tried out in testing. In multithreaded case, it is definitely worse than setting breakpoints on all relevant functions (functions cannot be traced without setting breakpoints at them in this case). In a single threaded case, depending on where the single breakpoint is set, it is possible to trace all code relevant to checking.

With MSVS in single threaded mode, line counts are the same for setting a breakpoint at only the start and at all functions except Idle; but if a breakpoint was set at Idle, line count dropped significantly. The reason is an Idle event was issued many times by the OS to the application, hence triggering the Idle event handler to run many times.

When running the same program in multi-threaded mode, the line counts stay the same across all runs at 40 and 30, for setting breakpoints at all functions including Idle and at start only respectively. When breakpoints are set at all functions excluding Idle, line counts varies significantly across runs, ranging from 40 to over 140, with an average being 73.35.

When junk threads are used and breakpoints set at all functions except Idle, line counts vary significant. Here we distinguish between useful lines (lines of code of threads doing useful work) and junk lines (lines of code from junk threads doing nothing useful). Numbers of junk threads tested were 2, 5, 10, 15, 20, and 25. When number of junk threads increases, lines of junk code increase and lines of useful code decreases overall.

The average for the various tests discussed above are plotted on the same graph in Figure 16. According to our test, when using multi-threading mode without junk threads, an average line count is about 73, compared to 113 in the single threaded mode. When junk threads are used, line counts for useful lines drop significantly even

if only 2 junk threads were used. As more junk threads are used, useful line counts drop even more. In contrast, junk line counts increase steadily at a slower pace as more junk threads are introduced.

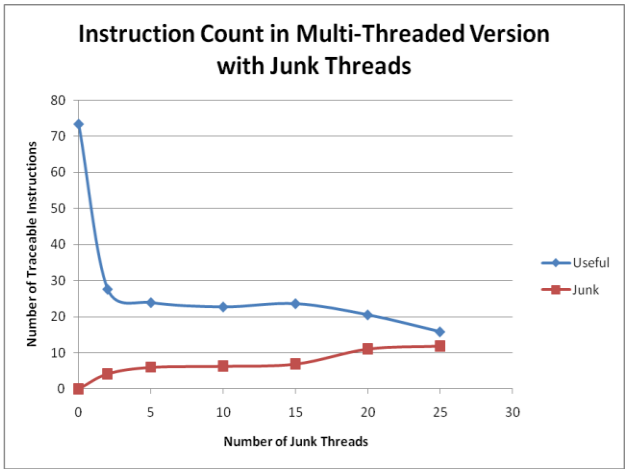

Figure 13. Average Line Counts of Useful and Junk Instructions

Figure 17 demonstrates a percentage count for the average number of traceable, useful instruction. As shown, only about 20% to 30% of useful instructions can be traced when junk threads are used, as opposed to about 75% when none are used. When 25 junk threads are used, traceable useful code drops to about 14%. From an attacker's perspective, the lower the percentage, the less useful code he can trace, which in turn means more difficult for the attacker to understand the code when it comes to reverse engineering.

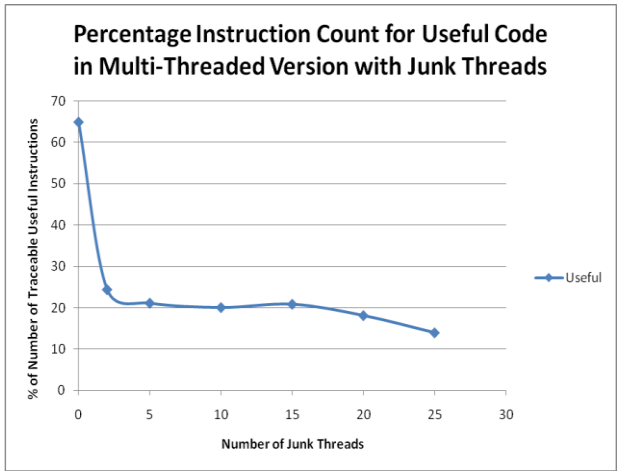

Figure 14. Traceable Useful Instructions

Next, we provide test results obtained using both OllyDbg [29] and IDA Pro [28]. First, tests were run with OllyDbg using the same pattern as with MSVS, with the exception that breakpoints were not set in the same way. Also, a different counting scheme is used. All of instruction counts were based on the calculation of addresses in blocks selected as relevant. Some of the codes were included in counts not executed by the debugger. They were only considered a rough estimate. Counts are likely to include a large number of instructions that are not relevant to checking; but rather, they are part of windows API libraries, such as the code executed initially to start the program or GUI libraries. High number of line counts is due to the inability to clearly identify relevant code correctly from disassembly. Because of the inability to identify code, no breakpoint is set in testing. An average result from the single threaded case will be called "total". When counting an instruction in a multi-threaded mode, a different approach is used. We will try to identify code that cannot be traced based on the thread table provided by OllyDbg, and subtract them away from the "total"; the resulting number will be regarded as the count for that particular test run. In theory, this number also represents the maximum amount of code an attacker can trace.

In single threaded mode, it seems like code can run normally; therefore, it should be theoretically possible to trace the code execution as long as breakpoints are smartly and properly set after correctly identifying relevant code sections in disassembly. Even though checking is done in a single thread, system still has other threads running in the background, such as the Idle event.

In a multithread mode, things get much more complicated:

- OllyDbg seems to capture the first available thread and executes that one in the foreground (making it available to step through). In this case, it appears to be always the same thread in our tests. Also, it appears like the thread captured by debugger is the runtime's GUI thread, which launches other checking threads. Once checking threads are launched, this captured thread pauses. Depending on how fast we step through this captured GUI thread, we may or may not see other threads because they can finish. In cases where we can jump into other threads, we cannot tell which checking thread (or even the Idle thread) we jump into.

- Repeated runs yield different results in our test runs. This can get even worse if we randomize the start order of checking threads.

- From running code in OllyDbg in multithreaded mode, we were unable to (or cannot easily) determine relationships among various threads (such as which depends on which).

- Setting breakpoints is an extremely difficult task in multithreaded mode, because one thread may block another. If we want to trace one thread, we must set a breakpoint for it. But if that thread runs in the background and it blocks the one running in the foreground (the one we are currently stepping through), then we will be in a deadlock like situation since the foreground

thread cannot proceed until the blocking thread finishes, which it cannot because of the breakpoint. If breakpoints are not properly used in a particular run, we cannot even bring up the GUI of the program (which happened quite often as we cannot set the breakpoints right). With no breakpoints set, we can get to the GUI of the program.

- While OllyDbg may take us to the code representing the thread (by double click on the thread), it is only possible when the thread has not finished execution. This may require one to work very fast.

Base on the results of test runs, it is clear that different execution paths are taken at different runs; therefore, resulting in a different count each time. Due to this fact, it is more difficult for an attacker to reverse engineer from the disassembly because he would get a different view of code each time he tries.

Testing with IDA Pro yielded similar results in single threaded mode as OllyDbg, although result is slightly different from that obtained from OllyDbg. This is due to inability to clearly identify code in the counting process. Overall, code seems to run OK in debugger, meaning it can be effectively traced and analyzed in theory.

In a multithread mode, things get much more complicated (even worse than OllyDbg):

- This time, we cannot even enter the password into the program, since it is launched in another thread. This is very devastating because without it, nothing else will run properly. IDA Pro clearly did not capture this thread in the foreground. This is going to be the end of it even if other code can run. We did not notice this before in OllyDbg since it got into a deadlock trap.

- IDA Pro, like OllyDbg seems to capture the first available thread it can and executes that one in the foreground (making it available to step through). In this case, it is appears to be always the same thread in our tests. But this thread it captured seems to run in an endless loop; it is perhaps the message processing thread from the runtime, or the Idle event thread. Even though it is able to show the different threads in a thread window, it cannot jump to any of them, not even to their location in disassembly.

- Again, we cannot determine relationships among various threads running code in IDA Pro in multithreaded mode, just like in OllyDbg, because we cannot step through them.

In the case of IDA Pro, we are not able to obtain a meaningful count of instructions in multi-threaded mode, because we cannot identify code corresponding to different threads. Testing with OllyDbg, these observations are obtained with junk threads added on top of other checking threads:

If junk threads are launched before checking threads, we were never able to get the program run correctly, as we got into the deadlock trap. No matter how many junk threads we used, the result was always the time, therefore, it is unclear whether number of junk threads matter because timing is another important factor. This is likely due to junk threads are launched before the useful checking threads (in a sequential order). When junk threads are launched first, they are the only threads around (in addition to system threads), and OllyDbg seems to capture one such thread (probably because checking threads are not even launched yet) and shows it in the foreground, and then wait indefinitely. In this case, two junk threads appears to be sufficient for our purpose.

If junk threads are launched after checking threads, the situation becomes more or less like the regular multithreaded case discussed earlier, except it is deadlock causing trouble instead of breakpoints (or actually can be both of them at the same time if breakpoints are set).

Assuming each thread, either junk or useful, has equal chance of being captured by OllyDbg and put to foreground, more junk thread should work to our benefits statistically in theory. Figure 18 shows the results obtained when various number of junk threads are used. In short, simple testing result on this can be generalized as the more junk threads the better.

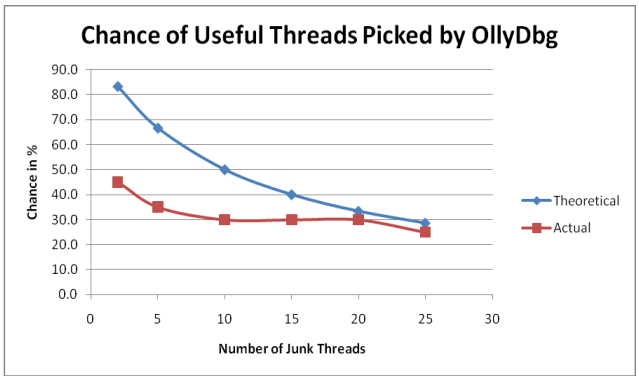

Figure 15. Useful Thread Selected

We are unable to repeat tests with IDA Pro, because we cannot identify code corresponding to threads. Because of this, we would tend to say from an attacker's prospective OllyDbg appears better than IDA Pro for purpose of reversing code.

To implement this new design using multiple threads, extra effort is needed. Extra efforts are summarized in Table 5.

Table 5. Effort Needed to Implement Proposed Design

Work	Development Effort
Dividing workload from single function into multiple smaller functions	This requires minimum effort, only a little extra time is required (about 30 minutes for this demo). This step is simple overall. Time is mostly spent on coding than analysis.
Ensuring dependencies among multiple threads are not changed	This requires some significant effort; about 2 extra hours are used. Time is mostly spent on analysis. In C#, about 40 lines of extra code are added for this purpose.
Coding the multiple threads	This requires minimum effort assuming one is familiar with the threading library in use. In this demo, about 10 minutes were needed for this part of coding. In C#, about 30 lines of extra code are added for this purpose.
Coding junk threads and deadlocks	This requires minimum effort. In this demo, about 5 minutes were needed for coding, and about 20 lines of code are written.
Other work related to multithreading	Coding timer function requires minimum effort, properly launching application in multithreaded mode also requires only little effort.

In summary, the extra effort in coding is not too difficult assuming one is already familiar with the library related to multithreading. On the other hand, making sure the design works properly requires more work in the analysis phase. In the demo, total effort is not more than 4 hours and approximately 100 extra lines of code; this is not much overall given the positive outcome.

Code obfuscation is also part of the new design; incorporation of it would make disassembled code more difficult to trace and force attackers to waste time by studying junk codes. In this project, XenoCode's built-in obfuscator was used primarily for this purpose. Obfuscation is achieved by inserting junk code into binary code. Without detailed analysis of its effects, the result seems good if the highest level of obfuscation is used. We plotted the effects of obfuscation of all 4 levels against the original source code, as shown in Figure 19. In Figure 19, each vertical bar is a comparison between the obfuscated code and the unobfuscated code. Areas colored in red represents a difference in code, whereas areas colored in white represents the same code. There are 4 levels of obfuscation provided by XenoCode, level 1 being the lightest obfuscation and level 4 being the heaviest obfuscation. The results in Figure 19 from left to right correspond to level 1 to level 4. As can be seen in the figure, there is very little white area at level 4, suggesting good obfuscation.

VII. Conclusions and Future Work

Our proposed design uses multiple threads and multiple validation modules for verifying serial numbers. After careful analysis of test results, running code in a multithreaded manner for checking serial numbers has clear advantages over the single threaded option. In particular, the following appear to be effective for our purpose:

1. Accepting user input in a thread other than the checking threads.
2. Running Idle event handler.
3. Use of junk thread and deadlock, especially launching them before useful ones.
4. Checking serial number in multiple threads.

Our method achieved the primary goal of this work. It proves cracking a serial number validation can be made more difficult if multiple threads are used instead of a single thread since it reduces the amount of traceable code. Also, overall extra efforts needed to implement the new design are small compared to that of the entire software development cycle, making this method practical to use.

We studied how multiple threads can make dynamic analysis of disassembly in debuggers more difficult to perform. Future research can be expanded to include how difficult it can be to extract code to create KeyGens from a multithreaded checking mechanism, especially when code is obfuscated by third party tools. Also, the effects of a running timer (especially those with short time intervals) could be studied further to understand its impact on debugging code. In addition, one could use third party tools to try to analyze interaction between threads to see if thread dependency can be found; and if so, can the dependency be understood. One could also try to use threads purposely running in an infinite loop instead of deadlocks to find out which method is better for our purpose. Finally, one can try to implement our new design in another programming language to see if our method still holds against attack.

References

[1] activatesoft.net, "Product Activation Overview", http://www.activatesoft.net/activation_overview.asp
[2] Chris Davies, "Windows 7 cracked after Lenovo OEM key leaks", http://www.slashgear.com/windows-7-cracked-after-lenovo-oem-key-leaks-2950684/
[3] ORC, "How to Crack", http://www.mindspring.com/~win32ch/howtocrk.zip
[4] Jianrui Zhang & Shengyu Li, "CS265 Project 2 Report", 05/11/2009
[5] MLC Technologies, "Hardware Key Activation", http://www.mcl-collection.com/support/licensing/hardware_key.php

[6] Schlumberger, "Cyberflex Access Cards Programmer's Guide", Jan 2004
[7] Bank of China, "Security Mechanism (Cooperate Service)", http://www.bankofchina.com/en/custserv/bocnet/200812/t20081212_144526.html
[8] Logic Protect, http://www.logicprotect.com/index.asp
[9] Mark Stamp, *Information Security: Principles and Practices*, Wiley 2006
[10] Mark Stamp, lecture notes on "Software Breaking", Fall 2009
[11] Wikipedia, http://en.wikipedia.org/wiki/Product_activation
[12] Martin Cowley, "Frontend Plush", http://frontend-plus.software.informer.com/
[13] Eric Lafortune, "ProGuard", http://proguard.sourceforge.net/
[14] Christian Collberg, "SandMark", http://sandmark.cs.arizona.edu/
[15] Scott Oaks, "Java Security", Published by O'Reilly, 2001
[16] Borland, JBuilder 2007 Documentation
[17] XenoCode, http://www.xenocode.com/
[18] Wikipedia, http://en.wikipedia.org/wiki/Polymorphic_code
[19] Shameen Akhter & Jason Roberts, "Multi-Core Programming: Increasing Performance through Software Multi-threading"
[20] BestSerials, http://www.bestserials.com/
[21] CrackLoader, http://www.crackloader.com/
[22] Australian Institute of Criminology, http://www.aic.gov.au/
[23] Jedisware, http://www.jedisware.com/
[24] Cyberlink, http://www.cyberlink.com/products/powerdvd/overview_en_US.html
[25] Chinmaan, http://i179.photobucket.com/albums/w306/chinmaan/activation.jpg
[26] RabLab, http://www.rarlab.com/
[27] Avast, http://www.avast.com/free-antivirus-download
[28] IDA Pro, http://www.hex-rays.com/idapro/
[29] OllyDbg, http://www.ollydbg.de/

Mark Stamp has worked in the field of information security for more than 20 years. Following academic work in cryptography, he spent seven years as a cryptanalyst with the National Security Agency, followed by two years developing a digital rights management product for a Silicon Valley startup company. For the past ten years, Dr. Stamp has been with the Computer Science at San Jose State University, where he teaches courses in information security. He has published numerous research articles and is the author of two textbooks, Information Security: Principles and Practice, 2nd edition (Wiley 2011) and Applied Cryptanalysis: Breaking Ciphers in the Real Word (Wiley 2007).

Jianrui (Louis) Zhang graduated from University of California, Berkeley with a BS degree in Electrical Engineering and Computer Science. Following graduation, he worked at Xignite, a startup company that provides financial data using web service technology. Louis then obtained a master's degree in Computer Science from San Jose State University, doing researching in software security. He is now working in the research department at Gilead Sciences, developing scientific software.