Abstract- Shareware programs are susceptible to piracy. Various reverse engineering methods used to circumvent software-based protection employed by existing shareware programs were analysed in this paper to provide insight into their weaknesses. These weaknesses include easily reverse-engineered algorithms, algorithms that compare the valid key with the entered key in plain text in memory, algorithms that reveal the actual key in memory and single point of failure. Because shareware programs are generally inexpensive and are developed by small teams, it may not be practical for shareware developers to implement protection solutions that are prohibitive in terms of cost and effort. A honeypot approach that can be implemented without significant effort, aimed at prolonging the time taken to fully circumvent the protection is proposed and tested. It has the following properties – appears to be easily defeated, is unpredictable and stealthy, and is a machine-dependent algorithm.

I. INTRODUCTION

Software piracy is a result of “unauthorised use, duplication, distribution or sale of commercially available software” [1], and is a significant economic problem. Software is susceptible to piracy because it can be duplicated at very little cost. The focus of this paper is on how shareware programmers can employ effective software-based protection techniques in relation to cost and development effort, to prevent piracy and maximise the return from their software. This paper also attempts to analyse the weaknesses of software-based protection schemes employed by shareware programs, and propose a honeypot approach that can be adopted by shareware programmers to deter the reverse engineering process. The reason for highlighting the process itself, instead of concentrating solely on specifics of the proposed approach is two-fold:

Firstly, shareware programmers can have many questions about the reverse engineering process and how it is possible to circumvent the protection without access to the original source code. Hence a comprehensive proposal without an analysis of the process could be confusing at best.

Secondly, repeatable reverse engineering methods could assist the shareware programming community in developing a structured approach to understand the inner-workings of software protection. In the reverse engineering process, programmers can appreciate a reverse engineer’s creativity and effort in circumventing the protection scheme that has been put in place.

II. LITERATURE REVIEW

A “honeypot” in the context of this paper is defined as a trap to detect and deflect attempts at reverse engineering software programs [15]. Honeypots have generally been associated with information systems and networks which are protected and isolated, which masquerade as source of value and interest to attackers. This approach when applied to software engineering has the objective of drawing reverse engineers away from the segment of codes that are protecting the software, into fake protection algorithms that appear to be easily defeated.

A survey of the literature on software-based protection exemplifies that most of the effort tends to concentrate on watermarking and obfuscation [2, 3, 4, 5, 6, 7, 8] and tamper-resistant software [9, 10, 11, 12]. Other methods that have been proposed include client-server architecture [13] and software aging [14]. The primary drawback of these approaches is whilst the techniques are claimed to be effective, they can be difficult to implement and may require significant development effort. The following briefly describes these related works.

A. Software watermarking and code obfuscation

Watermarking is a method used to embed secret structures into software to uniquely identify its developer or legal owner. For watermarking to be effective, it needs to be stealthy and resilient to dewatermarking attacks such as manual inspection by a human reverse engineer, code compilation, optimisation and obfuscation [4]. Nonetheless, software cost and performance are often the trade-offs for achieving these requirements.

Obfuscations can be simplified as code transformations that change the program structure aimed at confusing and deterring the reverse engineer, whilst preserving the functionality of the software [7]. Since obfuscation generally makes the code more difficult to comprehend, it may introduce other problems such as increasing the difficulty in the debugging process and adversely affecting program performance due to presence of redundant codes. Although it is possible to automate the process using commercial obfuscator on compiled executable files, incompatibilities and bugs can be introduced by the obfuscator because the developer would not have control over the methods used.

B. Tamper-resistant software

Tamper-resistant software is resilient to unauthorised modifications to its binary program code. This is commonly enforced using “checksum” and “repair” codes [10]. Like obfuscation, this can adversely affect program performance and introduce bugs even if the process can be automated, when not implemented carefully.
C. Client-server architecture

The method of dividing the program into an “open” component installed on an insecure machine (the client) and a “hidden” component on a secure machine (the server) is proposed in [13]. The “open” component is incomplete and only provides a subset of applications functionality. Hence, it is worthless to software pirates. An apparent downside to this is that if the server or the link over which interaction takes place fails, the software will cease to operate. This method would be more applicable where bandwidth for interaction and availability of the server is high.

D. Software aging

A technique was proposed in [14] whereby software updates are used to discourage piracy. Using this “software aging” method, both legitimate and illegitimate users are forced to update their software regularly in order to stay compatible with one another or benefit from new features/bug fixes present in the updates. In practice, the “software aging” method is widely employed by shareware developers. However, legitimate users can share the program updates and valid registration numbers with other illegitimate users via web forums and peer-to-peer networks, which hamper the effectiveness of this technique.

III. RESEARCH DESIGN AND METHODOLOGY

Software-based protection can generally be bypassed in two ways. One, by careful inspection of the disassembled code or program memory combined with reverse engineering of the algorithm to derive a valid registration key; and two, by making modification to the binary of applications functionality. Hence, it is worthless to software pirates. An apparent downside to this is that if the server or the link over which interaction takes place fails, the software will cease to operate. This method would be more applicable where bandwidth for interaction and availability of the server is high.

D. Software aging

A technique was proposed in [14] whereby software updates are used to discourage piracy. Using this “software aging” method, both legitimate and illegitimate users are forced to update their software regularly in order to stay compatible with one another or benefit from new features/bug fixes present in the updates. In practice, the “software aging” method is widely employed by shareware developers. However, legitimate users can share the program updates and valid registration numbers with other illegitimate users via web forums and peer-to-peer networks, which hamper the effectiveness of this technique.

To undertake an exploratory analysis into how this was possible without access to the program source code, a study was carried out on a random sample of 30 shareware applications written for the Microsoft Windows 2000 platform. The Windows operating system was chosen because it was the most widely deployed and targeted by software pirates at the time of study. Tools that were used to facilitate the analysis include kernel-mode and user-mode debuggers, disassembler, memory editor, registry and file system access monitors, and portable executable file protection scanner.

The virtual workstation software from VMware was used to provide hardware and operating system emulation. Primarily, this facilitated screenshots to be taken from the host operating system when a kernel mode debugger such as SoftICE was activated in the virtual operating system. Furthermore, VMware also allowed the backup and restore of the operating system in a matter of minutes. Each virtual operating system was saved as files located in the program’s subdirectory and was backed up by simply archiving the files. This was useful when analysing the behaviour of protection schemes. For example, an application could choose not to run if it detected the presence of a debugger. This could then be confirmed by running the application in a snapshot of a “clean” virtual operating system, without having to uninstall or re-install the debugger. In addition, if the operating system was corrupted during the reverse engineering process, VMware could be used to revert to a previously working copy of the operating system.

The random sample of shareware applications were installed in the virtual workstation. One-by-one, the protection scheme employed by the shareware were analysed and their general weaknesses observed.

IV. FINDINGS

Based on the exploratory analysis performed in Section III, the approach taken by a reverse engineer in circumventing the protection was observed and summarised in Fig. 1.

A. Find the starting point(s)

Given that a program contains multitude lines of codes, it is essential to pinpoint the exact location where the protection algorithm starts and ends. Although this seems impossible without access to the source code, it is not difficult in practice. Various tools are available to assist the reverse engineer, particularly debuggers, disassemblers and library function reference manuals. A debugger allows “tracing”, i.e. step-by-step execution of code. A disassembler translates machine code into assembly instructions, and reveals the library functions that are imported by the program. By knowing the library functions that are used by the program, the reverse engineer can quickly determine the starting points where the protection algorithms can be located.

Common library functions that are used as starting points include the MessageBoxA, GetWindowTextA and GetDlgItemTextA, all of which are exports from the USER32.DLL file. A common protection scheme prompts an error Message Box when an invalid registration number is entered. A debugger is used to set a breakpoint on the MessageBoxA library function to peek into the algorithm. In cases where the typical library functions are not imported, text string references can also be used. Other tools such as registry and file monitors are also used to determine the starting points. These tools can be used to locate exactly where the registration details are kept in the registry or file system.

B. Locate the weak point(s)

The ultimate goal of locating the starting points for reverse engineering is to be able to perform a white-box study of the program code, i.e. step-by-step examination of the protection algorithm. Debuggers are used to locate the offset where the library functions or text strings are referenced from the program. The crux of the protection algorithm can usually be found close-by for most shareware applications, for example a few instructions before or after these offsets. Debuggers can then be used to
step into and over the algorithm instructions to find out exactly what they do. A detailed understanding of the algorithm will enable the reverse engineer locate the weak points where the attacks can be targeted.

Where a white-box study is too time consuming or not possible, a black-box study can also be performed. This mainly involves the study of the input-output relations of the shareware application using tools with various functions such as registry and file monitoring, file scanning and memory editing. A memory editing tool is used to search for the memory residue of the valid registration number generated by the program’s protection algorithm. The program’s behaviour can be tested by modifying certain registry keys, which form the input to the protection algorithm. After the weak points are ascertained using these studies, the reverse engineer would typically choose the most efficient method in circumventing the protection, i.e. the method that requires the least effort.

C. Locate the weak point(s)

A software-based protection ultimate “crack” or breach is a “key generator” that can be used by anyone to generate valid registration number that turns the shareware into licensed and complete copies. This is often a result of an algorithm that can be easily reversed engineered, or an algorithm that makes a comparison of the correct key with the entered key in plain text in memory.

In the former scenario, a valid registration number is usually one that adheres to certain rules, and in rare cases, can be as simple as large number that is divisible by 3. The reverse engineer debugs through the program until the point where the algorithm can be found. With little effort, a “key generator” that reverses the algorithm is written. In other cases where the valid key is a function of the registered owner’s name, the algorithm can practically be copied instruction-for-instruction into another program that can be compiled into the “key generator” with minor modifications.

In the latter scenario, the reverse engineer traces through the program until the point where the comparison is made. The memory location that stores the correct registration key is determined. Next, using a debugger, the subsequent instructions are reassembled to print the correct code to the screen, for example using the MessageBoxA library function. This method can be used to defeat long and complicated key generation algorithms.

Other common weaknesses that are prone to attacks include “single point of failure” - a conditional jump that determines if the application is turned into a licensed copy or is still within the evaluation period, “memory residue” - algorithms that compare the correct key with the entered key in ASCII or UNICODE text are vulnerable to this attack, unless the memory content is erased immediately after the comparison is made, and “restorable evaluation period” - protection approach that allow full functionality of the program within a pre-defined evaluation period is susceptible to attack by having its data altered in Windows registry or on file.

V. PROPOSED HONEYPOT APPROACH

Any protection system can eventually be defeated. However, shareware programmers can work towards securing “a minimum length of time during which they could sell a large enough number of a newly released product” [10] using techniques that are practical in terms of cost and development effort.

A. Honeypot as psychological defence

Reverse engineers who target software protection schemes do it mainly for the challenge and the feeling of defeating a system better than others are able to. The more difficult a protection “seems”, the more it attracts attention and the more worthwhile it is to break the scheme. Because of this irony, solutions that boast strong protection are often defeated quickly. Hence, shareware programmers should make their protection scheme “appear” to be easily defeated, by using honeypot to trap the reverse engineers into believing that they have defeated the system. This includes fake protection algorithm that is activated whenever the presence of a debugger or disassembler is detected, using debugger and disassembler detection algorithms, combined with multilayer random checks and stealth. An algorithm that deceives the reverse engineer is more durable if it works in an unpredictable behaviour. In the case of a fake routine triggered by the detection of a debugger in memory, the fake key should only fail to work randomly. It should not display predictable behaviour, such as stopping to work after 30 days or whenever a program feature is used. Randomness impedes white-box analysis by making it more difficult for reverse engineers to pinpoint the algorithm. Similarly, for multilayer algorithm, the program should check for conformance to rules randomly. Whenever a violation of rule is detected, it should not terminate immediately or display any messages that indicate that a violation has occurred.

For fake and multilayer algorithms to work effectively, the keys must be a function of a machine dependent identifier. Although this is not a new approach, it prevents legitimate users from sharing “real” keys with other users. In defeating this, a reverse engineer would need to write a program that generates a valid key based on the machine identifier, or modify the algorithm in the program to accept any arbitrary key. Since fake and multilayer algorithms naturally aim to make both these effort more difficult, shareware programmers would benefit from the added protection when the combination of these techniques are used.

B. The Honeypot Approach

The honeypot approach is illustrated in Fig. 2, and has the following properties:

- **Appears to be easily defeated**: Decoy or multilayer algorithms are used to trick the reverse engineer into believing that the protection has been defeated.

- **Unpredictable and stealthy**: Exhibits random behaviour to hinder white-box analysis.
Machine-dependent algorithm: Keys are generated based on machine-dependent identifiers to render sharing of keys useless.

VI. EXPERIMENT, RESULTS AND DISCUSSIONS

To test the effectiveness of the proposed solution, a program, which incorporated the protection mechanisms proposed, was written using the Assembly language. The protected software was posted on a web forum frequented by reverse engineers. These reverse engineers were invited to break the protection without access to the neither the source code nor information about the scheme.

The “ShowDib2” bitmap viewer program was chosen as the basis for the rapid implementation of the protection system. This program was selected because the Assembly source code was readily available in the distribution of MASM32 assembler, the executable file was reasonably small (8KB) and hence manageable, there were sufficient features (e.g. Open, Save, Show) to implement multilayer checks, and made sufficient calls to known library functions to mask the calls made by the actual protection mechanisms.

To achieve the properties proposed in Section V, the original “ShowDib2” application was modified into the “Research_Crackme” program, using the logic and details depicted in Fig. 4, Fig. 5 and Table I. In order to unlock the program, a correct combination of “Registration Serial Number” and “Customer ID” was required to be added to the Windows Registry. Notable modifications are highlighted below:

- Included algorithms to detect kernel-mode and user-mode debuggers and disassemblers. Also detected other tools frequently used by reverse engineers such as registry and file monitors to find the “starting points”. Upon detection of these tools, only the fake algorithm would be activated.
- Included two bitmap files (“APOLLO11.BMP” and “PRINTER.BMP”), which would be displayed after the fake protection was defeated. This aimed to mislead reverse engineers who used these files when testing if they had broken the protection. Also, this was intended to mislead those who tested on “clean” operating systems that did not have debugging or disassembling tools installed.
- Included decoy text strings in the “.data” section of the program to lure reverse engineers into the honeypot, who use this method to find the “starting points” (Fig. 6). Text strings that would lead to the real protection were implemented as part of the program code so that they were not shown as referenced text (Fig. 7).
- Included random multilayer checks to validate the key used by the reverse engineer. Different parts of the algorithm were used to validate the key when starting the program, opening, changing the display and saving bitmap files. There was only 20% chance these different parts of the algorithm would be used to validate against the key used by the reverse engineer.
- The correct key was not revealed in memory. The key used by the reverse engineer was validated against a function (e.g. a factor of 5) rather than being compared to a text string.
- The real protection would not be defeated by modifying any particular conditional jumps because the validation was performed in a multilayer fashion, i.e. when a program feature such as saving bitmap files was requested.
- The fake algorithm stored data in a Windows registry branch where most reverse engineers would start looking (“HKLM\Software\Research_Crackme”). The real key on the other hand was stored in a branch shared by other programs (“HKLM\Software\Microsoft\Windows\SharedCom\DComCtl”). To avoid revealing the location of the real key, the honeypot would be activated if tools used to monitor access to the Windows registry were detected on the operating system.
Text strings meant as decoys were deliberately included in the “data” section of the program to lead the reverse engineer to the fake algorithm. This included the “HKLM\Software\Research_Crackme\Serial” string used in retrieving the Registration Serial Number from the Windows registry. Fig. 6 illustrates that this string was visible when analysed using a disassembler.

To increase stealth, text strings such as those used in detecting debuggers and storing the real key in the Windows Registry were implemented as part of the program code so that they would not be shown as referenced text. An example of this implementation is depicted in Fig. 7.
Participants were also requested to complete a questionnaire, including the rating of their reverse engineering competency into categories of “Amateur”, “Intermediate” and “Expert”. This was based on criteria such as number of years spent in reverse engineering software protection and their skill sets e.g. ability to write “key generators”. If the protection was not broken at first attempt, the participant was informed and asked to retry. This was repeated until either the participant had given up or succeeded. The number of attempts made before succeeding was recorded.

A total of 12 reverse engineers, comprising 4 “Amateurs”, 6 “Intermediates”, and 2 “Experts” participated in the experiment. The following observations were made:

- All participants rated the protection as “Easy”, but none provided a completely working solution at first attempt.

- Only one “Intermediate” participant provided a completely working “key generator” on the third attempt.

- Three participants (one from each competency rating) provided serial numbers that only defeated the fake algorithm.

- The remaining eight participants provided patches; only 3 were completely working. Of the three working solutions, two were submitted by “Intermediate” participants on the third attempt, and one by “Amateur” participant on the sixth attempt.

- In the end, eight participants comprising 3 “Amateurs”, 3 “Intermediates” and 2 “Experts” opted-out from the experiment on the reason that it would have taken them too much time to provide a working solution.

In general, the honeypot approach was effective in misleading all the participants in the experiment. Specifically, two participants refused to accept that their solution did not work. Debugging tools were detected their on the operating systems, which activated the fake algorithm to accept the fake key. Another participant was confused why a registration serial number that was thought to be the correct key did not work occasionally. Other participants were able to discover the hidden checks after being told that their original solution did not work. One participant managed to monitor accesses to the Windows registry by defeating the detection algorithm, but overlooked the “HKLM\Software\Microsoft\Windows\SharedCom\DComCtl” branch where the real key was supposed to be located. The participant thought that had something to do with COM controls.

In practice, the mechanisms proposed can be effective because they take advantage of the fact that most “crackers” do not spend a lot of time testing their solution thoroughly. Moreover, when the mechanisms behave in an unpredictable and stealthy manner, the time taken to fully circumvent the system is further prolonged.

VII. CONCLUSIONS

Due to the characteristics of shareware applications that are generally inexpensive and shareware development teams that are small, it may not be practical for shareware developers to implement protection solutions that are prohibitive in terms of cost and effort. This paper hence put forward a software-based protection approach that can be implemented without significant effort by shareware developers.

To understand the rationale behind this approach, various reverse engineering methods used to circumvent schemes employed by many shareware applications were discussed. The analysis of this process provided insight into the weaknesses of these schemes. The common weaknesses shared by shareware applications include easily reverse-engineered algorithms, algorithms that compares the valid key with the entered key in plain text in memory, algorithms that reveal the actual key in memory, and single point of failure e.g. scheme that can be defeated by replacing the conditional jump with an unconditional jump instruction.

From these weaknesses, a honeypot approach aimed at prolonging the time taken to fully circumvent the protection has been proposed and tested. It has the following properties – appears to be easily defeated, is unpredictable and stealthy, and is a machine-dependent algorithm.

Because this approach suggests unpredictable and stealthy program behaviour when an illegitimate key is used, unauthorised users may interpret this as a bug. One way of isolating bugs from intended behaviour is to catch all exceptions and write them to a log file. These exceptions nonetheless should ideally exclude registration key violations; otherwise reverse engineers could use them as starting points to locate the algorithm. In practice, shareware developers usually ask for user registration keys when providing technical support. Problems faced by legitimate users would most likely indicate bugs within the program functionality than intended behaviour.

VIII. RECOMMENDATIONS

Future research work should focus on exactly how the proposed approach can be effectively implemented by shareware programmers, how they may be vulnerable to other attacks and to what extent they
can be improved. Statistical research can also be conducted to determine the types of protection employed by shareware programs and their effectiveness in practice, e.g. the speed in number of days to which a “key generator”, “patch” or valid key is made available for that program after it is released to the public.

REFERENCES


