Towards More Effective Virus Detectors

Raghunathan Srinivasan and Partha Dasgupta
Arizona State University

1. Introduction

Viruses (or malware) are a scourge, with potentially unlimited fraudulent uses. Smart viruses can hide, mutate and disable detection methods. Computers are an important part of everyday life to many people across the world. The Internet has revolutionized everyday life. The Internet has also brought an ugly side of computers: a plethora of malware. Home computers are most vulnerable to attacks by malicious programs and hackers. This is because many home users are less equipped to prevent or counter an infection. Even if the user possesses the required skills, a smart virus that appropriately hooks onto the system can hide its presence on the machine, and remain undetected. These compromised machines are vulnerable to hackers who steal secret data or even install additional software that enables the use of the machine as part of a *botnet* to launch Denial of Service attacks on servers, or to intrude on government agencies.

Virus writers use a variety of techniques to attack a machine. They can be enumerated as follows:

- Social engineering
- Spamming
- Exploitation of software vulnerabilities
- Code Injection
- Cross Site Scripting
- Pharming

Elimination of software vulnerabilities requires the implementation of secure OS and secure coding. Both the issues have been researched heavily but have been ineffective in practice, mainly due to the abundance of legacy code. The OS kernel consists of millions of lines of code, and writing a secure OS would require that the entire kernel is bug free. Writing bug free code is a very complex problem. Creation of a completely secure OS is unlikely (Basili & Perricone, 1984). The problem of preventing infections is made difficult by the fact that most hackers rely on human error (social engineering) to compromise systems. It can be inferred from above that it is hard to prevent an infection since it is difficult to foresee the exact error a user may commit. Hence, security software rely on detection instead of prevention.

Software such as Anti-Virus (AV) solutions, and firewalls offer some protection against computer attacks; however, they are not completely effective. Virus detection is surprisingly hard, it has been shown that there is no algorithm that can perfectly detect the presence of malicious code (Cohen, 1993). Since the AV relies on definitions or known behavioural patterns of malicious programs, a code that is new in design can effectively use the *zero day exploit* (Schneier, 2003).

The AV and other security software suffer from several shortcomings. The AV is a user level application that can be killed by any process with administrator privileges, or it can be infected by viruses, due to which the detection engine is rendered useless. Like a virus, the AV software may attempt to hide itself, but such attempts to hide can also be detected.

Software in most machines is identical (genetic uniformity). Due to this, an attacker can use one machine to carry out experiments and find out ways to exploit vulnerabilities, and use the information to carry out the same attack on other machines. By making programs dissimilar on every machine the complexity and cost of an attack can be increased.

Motivations behind malwares have changed constantly over time. Early viruses were designed to cause disruptions by wiping out hard drives and deleting files. Recent malwares are aimed at stealing information such as bank account numbers, credit card information. The payload of a malware has also undergone changes. It may contain a virus, rootkit and a password logger. Malwares are a big threat in today's computing world.

AV software has evolved continuously with malware (Nachenberg, 1997; Sanok, 2005). AV products have made it tougher for viruses to escape detection. The virus writers have responded by creating a new trend. Malicious programs disable the AV and other security related processes in the system.

The *SpamThru* Trojan gets installed on a host system by social engineering. It patches the running AV to block updates and prevent its detection. It installs a pirated and patched copy of a popular AV to scan the system to remove other malwares. This is done to ensure that there are no competitors for system resources. It runs a root kit to conceal its own files from the scanner and system (Naraine, 2006). *Beast* is a backdoor Trojan horse; it works as a Remote Administration Tool. It injects its DLL's into explorer and winlogon. Once it infects a system, it shuts off the AV, Firewall, and the attacker obtains control of the system (The Beast, (n.d.)).

This list is not limited to only these two; Klez, Bugbear and Lirva are other examples of viruses that disable AV programs. This is known as *Armoring* (Chen, 2003). Armoring marks a significant change in virus behaviour. Till now any infection could be

contained and cleaned by the AV after the arrival of an update, however, the latest trend of killing the AV process threatens to make their presence inconsequential. This means that there is an urgent necessity to protect the AV from rogue programs.

This paper presents a software based solution to prevent malware from disabling security software. This problem is similar to that of preventing infections and also similar to the problems faced by virus writers in hiding their programs from the AV. It is not possible to provide a solution that will hide the AV from a malware completely; however, this paper aims to make the process of locating and killing the AV difficult.

2. Related Work

Hiding information is used for malicious and benevolent purposes. The benevolent uses are to hide passwords, credit card information and code obfuscation for DRM. Malicious uses are typically to hide the presence of malware. To achieve this, the malware monitors and intercepts the state and actions of the compromised system. A *Rootkit* is a popular tool used by hackers to hide the presence of malicious entities in the system. *Shadow Walker* (Sparks & Butler, 2005) is a rootkit designed to deceive in memory signature scanners. It hooks on to the page fault handler and the page table entries in the system. It detects the read requests made by the scanners and provides fake values for the corrupted section of memory to remain hidden. *SubVirt* (King & Chen, 2006) and *Bluepill* (Rutkowska, 2006) are Virtual Machine (VM) based rootkits that take advantage of the fact that the lower layers in a system can effectively control the upper layers. SubVirt and Bluepill install themselves between the hardware and the operating system to control the machine. These rootkits cannot be detected by processes running within the system. The exact sequence of events in the installation process for the rootkits is beyond the scope of this paper.

It can be seen that use of a rootkit ensures that a process remains hidden in the system from other system programs, hence may be used to hide the AV in the system. However, the problem with this approach is that if in any eventuality a virus patches on to the AV software then the virus can never be removed, also the aim of this paper is to hide the AV from malicious code, and not the system administrator.

Another reason for not using any approach similar to rootkits is that it would involve placing the AV inside the kernel of the OS. The AV requires frequent updates. Updating the kernel or a VM is a tedious process; hence, the AV process must remain as an application in the user space.

Code Injection is a technique used to introduce code into a process from an outside source during execution. These techniques are very popular in system hacking and cracking. Kc, Keromytis and Prevelakis (2003) describe code injection methodologies for various languages and platforms. Benevolent use of code injection occurs when a user changes the behaviour of a program to meet system requirements. This is done when the cost of modifying the software is a costly process and it is cheaper and convenient to inject code in the program to achieve the desired functionality. In this paper, code injection is used as one of the means to hide the AV process in the system.

3. Threat Model

All security related problems cannot have a single universal solution. Each solution lives up to a threat model. A threat model describes the assumptions and factors considered while making a solution. It also describes the problems that are addressed by the solution. The assumptions made in this paper are: The AV will get installed on a clean machine. The virus will not attempt to kill all processes, or delete all files in the system. The virus will allow some application to upgrade to newer versions. Rootkits are not installed on the system. This solution works effectively against malware that attempt to identify the AV by scanning the system registry, process table entries and file system for the presence of known AV software solutions. This solution also works effectively against programs that identify the AV by the files and libraries used by it.

4. Design

To evade detection by malicious programs, the AV should remain hidden from all processes in the system. The reason for this is that any program on the machine may be infected. To effectively hide a program, its file structure, registry entries and process table entries have to be hidden. These issues are addressed by a two fold process. The first step involves installing the program as a different program on the machine. This serves to hide the file structure and registry entries, and also ensures each copy of the AV looks different. The next step involves using code injection to migrate the program code and library into other processes. Migration of code serves to hide process table entries from all other system components. By performing code injection and the subsequent migration after certain time intervals, another threat is addressed. It becomes difficult for malware to locate where the AV resides currently even if it finds where the AV resided previously.

The design of the solution is illustrated in Figure 1; this solution was implemented on the Windows 2000 platform.

4 a) Installing the Program

Viruses are known to insert sections of their code in other programs to hide their presence. A similar trick can be used to hide the AV. Writing part of the AV code on an executable is not a good solution as it would be too much virus like. Instead, the AV is installed as a different program. This involves replicating the directory structure and file names of the software being replicated. The installation suite contains the list of commonly used software in consumer computers. During installation, the suite finds out the software in the list have not been installed on the machine. The suite then provides the truncated list to the user to choose the software in whose name and structure the AV should be installed. On obtaining the response, the suite proceeds to replicate the directory, file structure and registry entries of the chosen software. By obtaining user response, the solution ensures that the name and directory structure of the AV is different in every user machine. This provides the genetic diversity that helps in cloaking the AV system.

4 b) Starting the Process

The first step in hiding the AV is to cloak the point from where the process loads. Malware search registry entries to find values that match the names of popular AV software. The registry entry containing information about the location of start up items is vulnerable to attacks; hence this entry has to be cloaked or removed. This is achieved by forcing another process to start the AV. The best choices for the starter process are system programs that load on boot.

This part of the solution was implemented by inserting a call to load the AV program inside the code of msgina.dll, a library used by the system process *winlogon*. If this process is different in every machine, then it would be very difficult for a malware to detect where the start up information of the AV is stored.

4 c) Execution of the Process

In the previous two sections, it has been made fairly difficult for malware to identify and disable the AV; however, there exists a threat that a program may identify the AV by taking the snapshot of the system at any given time and analyse the result to identify the AV. To make it tougher for the malware to disable the AV, code injection is used to move the AV code and libraries from one process space to another.

To achieve this, the scheme described by Kuster, R (2003) to inject code and library into another process. The user is requested to enter a random sequence every time the machine boots. The AV process chooses a target process running in the system using the entered value after time period ‘x’. Once this process is chosen, the libraries and code are injected into it. This process occurs after every ‘x’ period of time, it must be noted that ‘x’ is a value that can be set by the system administrator on every system.

4 d) Watch Processes

Malicious programs run a system query to identify the AV process. The same technique is used to monitor whether the Anti virus is running on the system or not. A standalone process can monitor whether the AV is disabled, or for better results, ‘N’ different processes can monitor the AV. Each of these monitors the AV process by receiving the name of the AV and the random sequence provided by the user as a start up parameter. These processes locate the AV program in the injected processes with the aid of the random sequence, and restart the program with human supervision in case the AV is disabled. In addition, each process also receives the name of the other ‘N-1’ processes so that every watch process can be monitored. The watch processes also compute and store the hash values of the known good copy of the installed AV software and the modified system library files. Prior to shutdown, the watch processes check if any files have been modified, if so, the user is notified to perform a re-installation of the AV.

This was implemented by using 3 processes to monitor the AV. Each process calls the system API *GetProcessId* to find whether the AV and the other watch processes are executing. If a watch process is disabled, then it is started immediately. If the AV is disabled, then the user is prompted to start the AV. If the user declines to start the program, the answer is stored in memory to avoid prompting at a later time.

It can be argued that a malware may store the integrity values of all known software, binaries and libraries, and compare these values with the files in a target system to identify the possible presence of the AV. However, the size of such a database would be very large and computing results would require extremely high storage and computational complexity. A malware is typically a light-weight program that is designed to work without catching the user’s attention; hence, this technique would be infeasible. This issue is also partially solved by making the watch threads perform integrity check during system shut-down.

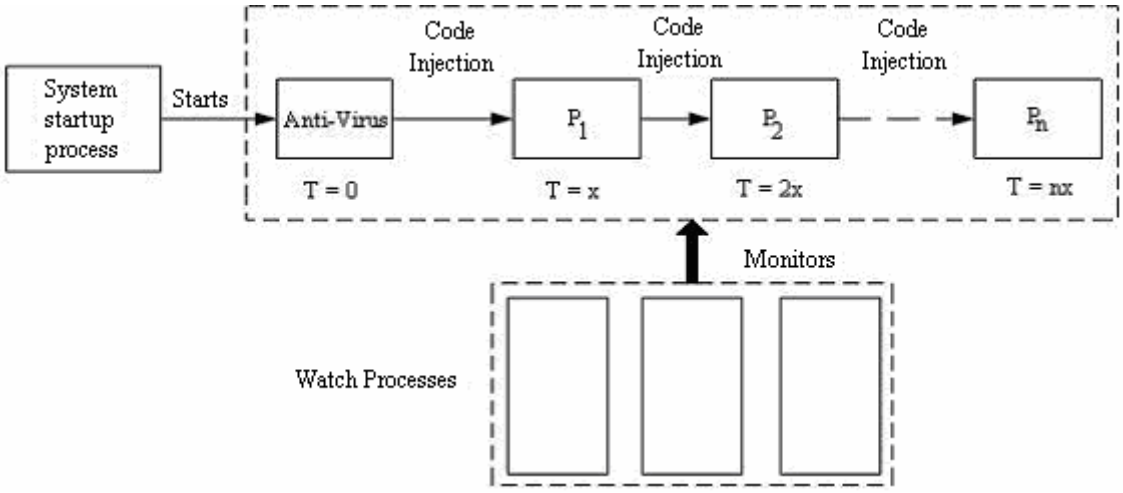

*Figure*1: Design for hiding the Anti-virus from malware

5. Conclusion and Future Work

This paper highlighted the growing problem of malicious programs disabling the security software and the need to tackle it. A software based solution was presented to hide the AV program in the system from malware. The solution provided protection from malware that scan the registry entries, file structure, and process table entries for the presence of the AV by installing it as a different program and cloaking its start up information. The solution also provided migration of code to counter malware that may attack the AV program by taking a system snapshot and computing offline results. Finally, multiple watch processes were introduced to monitor the AV and perform some shut down events that are critical to maintaining the integrity of the AV.

As seen in section 1, most malware successfully use the zero day exploit. The reason for this is that AV uses *Blacklists* to identify malicious code. If AV solutions migrate to using a list of known good programs (*White-list*), then the zero day exploit can be countered and many viral infections can be prevented. The only argument against usage of white-lists is that there are too many good programs around. However, all of them are not likely to reside on every system. The AV program can scan the system on installation to store a white-list. Every time a new program is detected on the machine, the user can be prompted to identify it. If the user cannot identify the program, it can be discarded or quarantined. A combination of white-lists and blacklists can serve to make consumer computing secure, and should be incorporated in Anti-virus solutions.

References

Basili, V.R. and Perricone, B.T. (1984). Software errors and complexity: an empirical investigation 0. *Communications of the ACM*. 27, 42 – 52.

Cohen, F.B. (1993). Operating system protection through program evolution. *Computers and Security*, 12. 565 – 584

Schneier, B. (2003). Attack trends: 2004 and 2005. *Q focus: security*, 3(5). 52 - 53.

Nachenberg, C. (1997). Computer virus-antivirus coevolution, *Communications of the ACM*, 40. 46 - 51

Sanok. Jr, D.J. (2005), An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. *InfoSecCD '05*, 142 - 144

Naraine, R. (2006). Spam Trojan Installs Own Anti-Virus Scanner. Retrieved October 20, 2006. Website:http://www.eweek.com/article2/0,1895,2034680,00.asp

The Beast. (n.d.). Retrieved October 13, 2005. Website:http://lists.virus.org/dshield-0310/msg00337.html

Chen, T.M. (2003). Trends in Viruses and Worms. *The Internet Protocol Journal*, 6(3). 23 - 33

Sparks, S., & Butler, J. (2005). Shadow Walker: Raising the bar for windows rootkit detection. *Black Hat*.

King, S.T., & Chen, P.M. (2006). SubVirt: implementing malware with virtual machines. *Security and Privacy, IEEE*, pp 14 – 28.

Rutkowska, J. (2006). Subverting Vista Kernel for Fun and Profit. *Black Hat*.

Kc, G.S., Keromytis, A.D., and Prevelakis, V. (2003). Countering Code-Injection Attacks With Instruction-Set Randomization. *ACM CCS*, 272 – 280.

Kuster, R. (2003). Three ways to inject your code into another process. *www.codeproject.com*