Evasive Security Using ACLs on Threads

Using firewall-like rules to prevent malware

Saifur Rahman Mohsin
Executive
Raaj Construction
Trichy, Tamil Nadu, India

Abstract*— This document describes a new architecture for security systems, which greatly improves system performance and at the same time enforces veritable security to it. Conventional anti-virus systems that exist in the market today consume massive amount of memory (both physical as well as permanent) and cannot be relied upon by power users who need to tap into the full potential of their devices. Also, these systems cannot be employed in thin clients, which have limitations in quality and capacity of hardware. The document describes a better architecture for the detection and prevention of malware with an escalated improvement in overall performance of the system. It also discusses new concepts that may be ported into existing security systems to improve the efficiency of those systems.

***Index Terms*—Computer Security, Just-in-time Detection, Malware Prevention, Security Architecture**

I. Introduction

Antivirus systems have been in use for a long time now since the numbers of virus has been increasing in the past few years. Different anti-virus systems provide a variety of distinctive features that offer to make systems highly secure from malware. However, the fundamental way they function is always consistent. Conventional anti-viruses scan recursively burrowing into file systems as well as connected peripherals identifying files that contain malware signatures either in plain text or obfuscated formats. The problem is that this process is a high energy consuming process and also requires a bulk of memory in order to function. These systems also require to be updated frequently to update the malware definitions (i.e. the signatures) of new malware that are created every hour.

This paper describes a more efficient architecture to overcome the problems of the existing conventional systems. This is done by looking at malware in their most primitive form to understand the basic process that any malware involves itself in--from hooking itself into the operating system to executing stealthily. We also look at as how anti-virus systems detect these malicious files so that we can create a more intuitive and real-time system that handles malware effectively before it attacks the system.

II. How Viruses Work

Malware come in many different forms. It ranges from commons viruses, to malicious scripts, worms, Trojans, rootkits, etc. A virus is anything that can cause havoc to the confidentiality, integrity or availability of a system.

Regardless of whether a given malware is a virus, worm, or Trojan, it always requires a thread to execute. Like any program, it consists of several code statements and also has a single point of entry to execute. This means that like any program, a virus requests the operating system for memory and processor cycles (as shown in Fig. 1) to execute. It also contains a process table and is associated with a Process ID (pid) and handle ID.

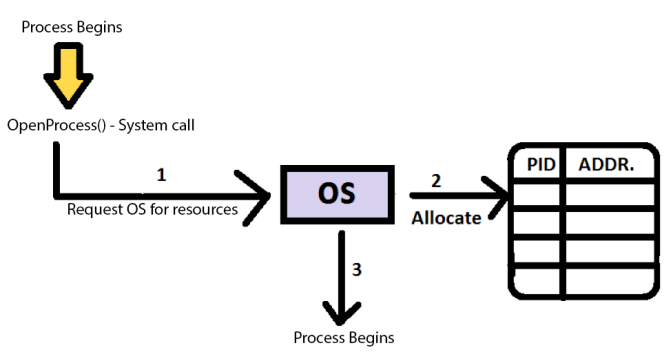

Fig. 1. How a process begins to execute

Most viruses are targeted towards a certain kind of resource. The intended resource may range from private information, confidential data, illegal use of computational power, or even to serve ads (in case of adware). The code that is written in a virus program must be well hidden from the anti-virus system as well as other programmers who might reverse engineer the code for their own deeds. A good virus therefore is encrypted in a format that makes it hard for anti-virus systems to neither detect nor reverse engineering to be feasible. This process is known as *obfuscation* and most good viruses that are existent today are well obfuscated.

In addition to obfuscation, most viruses are attached to legitimate files so that they are more concealed. There is a higher chance of a virus being installed as a sub component of an infected program rather than as an individual program itself. This ensures that the virus program does not appear to be malicious thereby preventing anti-virus (AV) systems to flag it. Such a program that is highly invisible to an AV is considered to be Fully UnDetectable (FUD). There are several FUD viruses in existence today. These are no longer just

targeted towards computers but are aimed towards national infrastructure like power grids, nuclear plants, automation plants, etc. Therefore, the conventional AV systems do not provide the necessary security that is required and hence we must look for better solutions.

Our goal is to develop such a system and which detects and informs the end user about the existence of these new threads (or processes) which the user never intended to execute. A user may decide whether he really wants to execute a given process and the system can mark these choices as trusted processes (in a similar fashion as websites are marked as trusted by firewalls and browsers). Everything else can be blocked (these will include the threads created by malicious programs), as it seems as an unnecessary overhead for the system to execute these processes regardless of whether these are malicious or not.

III. How Anti-Virus Software Works

There are several AVs available in the market today. A lot of them offer special features, which have a certain edge over other systems. Regardless of these features, a typical AV system consists of few common components – A database of virus signatures, a scanner and few auxiliary modules to make it easy for the user to customize the way the AV system behaves. A definition repository (a file or database) contains several code signatures that may be used to identify if a file is infected / infectious. These signatures need to be frequently updated as time progresses and new viruses are manufactured every hour. This becomes an overhead to the user to update the system every now and then. Also, the AV system tends to become larger and larger as time progresses due to the vastness of the definition file and the enormity of the new signatures that keep getting added. To complicate this further, the AV system itself contains a scanner module, which sweeps through the file system, testing each file if it may contain plain text or obfuscated forms of the virus signature. The entire search process requires a substantial amount of memory and slows down the overall performance of the system. It has been observed that a system is much faster when an AV system is not present than otherwise.

Thus, it is possible to conclude that an anti-virus system, which uses the conventional architecture, cannot be a real-time system and the success rate and efficiency depends on the number of virus signatures present as well as the kind of obfuscation algorithms that it can detect. Our goal is to describe a better architecture that prevents this overhead and also serves a higher success rate. Such a system is described in the next section.

IV. The Improved Architecture

It has already been described in Section II that any malware requires a space in the process table in order to run and therefore has an entry point. Exploiting this fact, we can design an architecture that ensures that each process that starts must be filtered using a set of rules that can be pre-defined as well as user-defined. This is very much alike to the system used by “firewalls”--where a list is used to decide whether a website / hostname must be blocked or allowed. Such a list specifies the limitation of the access and is known as an Access Control List (ACL). We implement this concept into anti-virus detection systems to realize a more effective architecture.

The efficiency of this lies in the fact that there is no scanning mechanism to search for viruses. Instead, the detection is done just at the moment a program requests the OS (As shown in Fig. 2) to provide resources for it. This means that we are performing a Just-In-Time Detection (JIT-D) and that is why this architecture is more effective compared to the existing ones that are in use today.

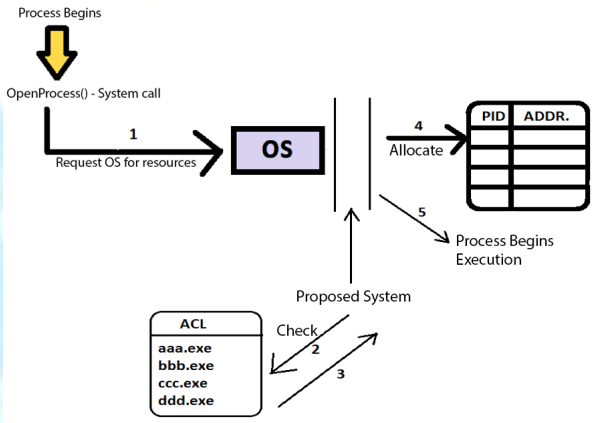

Fig. 2. How the proposed system intervenes process startup

In order to increase the efficiency much further, we need not rescan a process if it has already been scanned. However, this causes a risk because a user may decide to update the clean file with a malicious one. For this reason, we take the hash of the file contents and store it in the ACL along with the process name. There is no need to store the path as the hash ensures that a particular file is always uniquely identified. Every time the process starts our system will ensure that the present hash is identical to the stored hash in ACL. If so, it will not scan the file. If not, it should scan the file as it has been altered. This technique is highly effective because it is extremely difficult even for any skilled programmer to write a malware program whose hash value exactly coincides with an existing program.

V. Access Control Lists

ACLs have been in use in operating systems for decades. They are generally used to specify file rights to provide authority for who has access to a particular file. An ACL may be a simple text file that contains mappings of process names along with their permissions and hashes.

Processes that start are first checked as to whether an entry for the process exists in the ACL. If it does, then it is checked whether it has been modified using the computed hash. If it does not match, the anti-malware system is invoked to process the initiated process and determine whether it is safe to execute or not.

We use a standard hash like MD5 or SHA-1 to uniquely identify the file. The reason being that some files tend to change when a user updates the file by replacing it. The architecture explained her strongly enforces that files /

programs need not be processed unless its data has been modified. Hence, by using a computed hash we can easily detect changes.

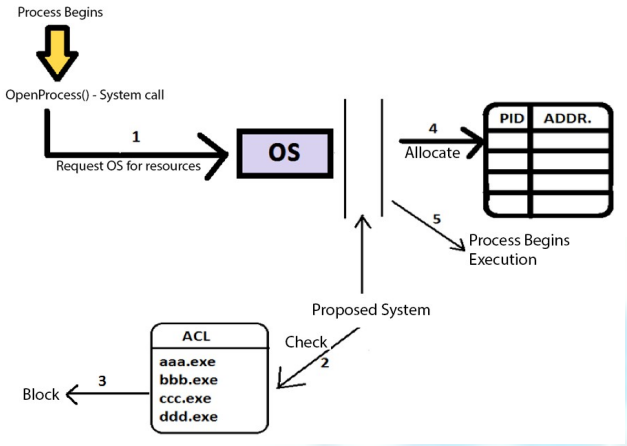

Fig. 3. How the proposed system blocks malicious processes

An ACL therefore contains program names, their computed hashes and the access levels. If upon execution of a new process, the process exists in ACL and is marked as a blocked process then the system prevents the execution of the process (See Fig. 3) and this prevents malicious code from infecting the system.

VI. Just-In-Time Detection

It has already been well explained in Section III that most AVs are pretty slow. They not only take a lot of time to run but also consumes a large amount of physical memory. This causes a lot of lag in other programs and slows down the overall performance of the system.

By detecting the processes exactly at the time of their execution, we remove the need for a scanner to traverse through the hard drives as well as connected peripherals. It enables the program to run at a very low memory cost and high efficiency with momentary bursts of memory usage when a process is caught. We call this technique as Just-in-time detection simply because it detects just before the time of execution. This means that it is absolutely allowed for a malicious file to sit in a hard drive as long as it's not harmful. The moment it is executed, our system kicks in to prevent it from causing harm to the system.

VII. Implementation

Using C# an implementation was made using the Diagnosis namespace in the .NET framework to intercept process initiation. By suspending the thread until the process completes, it was possible to prevent the system to get infected in processing time itself. Also, it was identified that this architecture is self-preserving as it prevented even the ACLs to be directly modified by a process. The implementation was targeted towards the Windows operating system, as it is the most widely used system and has more viruses when compared to other systems. An analysis was also taken (As shown in Fig. 4) which determined that our system was highly efficient in memory consumption as it was comparatively nothing to other AV systems or even other programs. Hence, it was determined that this architecture is the best approach for malware prevention as it saves a lot of resources.

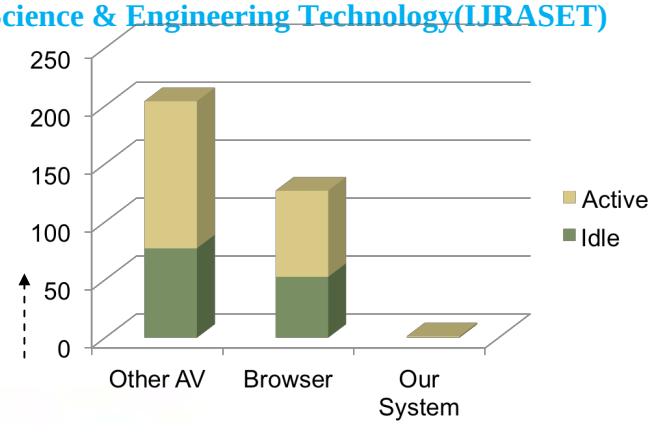

Fig. 4. Memory usage statistics of the proposed system against other applications—browser and another AV system

References

[1] Fred B. Schneider, Cornell University "Least Privilege and More" [IEEE Computer Society 1540-7993/03, 2003].

[2] "Managing Authorization and Access Control". Microsoft Technet. 2005-11-03. Retrieved 2013-04-08.

[3] Cynthia E. Irvine, Naval Postgraduate School "Teaching Constructive Security" [IEEE Computer Society 1540-7993/03, 2003 under Security & Privacy].

[4] "Access Control Lists". MSDN Library. 2012-10-26. Retrieved 2013-04-08.

[5] Elias Levy, Architect & former CTO, Symantec "Approaching Zero" [IEEE Computer Society 1540-7993, 2003 under Attack Trends, 2004].