

External Monitoring of Endpoint Configuration Compliance

Darrell M. Kienzle, Ryan K. Persaud, and Matthew C. Elder

Symantec Research Labs
Symantec Corporation
2350 Corporate Park Dr. #300
Herndon, VA 20171

{ darrell_kienzle | ryan_persaud | matthew_elder }@symantec.com

ABSTRACT

We describe a system for externally monitoring endpoint configuration compliance of an end user system that provides a high assurance monitoring function and data. Typical approaches to monitoring for endpoint configuration compliance rely on the integrity of the endpoint's operating system and do not protect the monitoring function from subversion or spoofing by threats from within the monitored system. Our approach utilizes (1) a virtual machine architecture on the endpoint system to protect the monitoring function and (2) virtual machine introspection of the end user's environment. In this paper we describe our approach to external monitoring of endpoint configuration compliance, present the technical details of our monitoring system, and discuss some of the issues associated with external monitoring.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—*invasive software*; K.6.5 [Management of Computing and Information]: Security and Protection—*invasive software*

General Terms

Security.

Keywords

Configuration compliance, security, trusted computing, virtualization.

1. INTRODUCTION

Endpoint configuration compliance management ensures that each client machine on an enterprise network complies with an organization's configuration and security management policies. Typical configuration compliance policies prescribe approved and prohibited hardware, operating systems, and applications (including the appropriate version levels, updates, patches and settings), as well as the installation and operation of security software utilizing the up-to-date information regarding vulnerabilities and malware of concern. These configuration compliance policies are often derived from system administration and security "best practices" documents and/or formal industry or

regulatory frameworks such as the Federal Information Security Management Act (FISMA) and the Health Insurance Portability and Accountability Act (HIPAA). Appropriate configuration compliance reduces both system administration costs and exposure to system compromise from poor system administration, misconfiguration, malicious code, and other vulnerabilities.

Configuration compliance monitoring typically relies on the integrity of the monitored system's operating system (OS) and application software to report accurate and complete information regarding an endpoint's configuration. Unfortunately, this means that configuration compliance reporting is susceptible to spoofing and/or subversion from attackers and malicious code. Existing approaches to configuration compliance monitoring either have no visibility into the endpoint system's integrity from over the network, or the monitoring agent is subjected to the same threats as the monitored system and therefore susceptible to subversion, tampering, and/or compromise.

We propose a method of monitoring for security and configuration compliance that provides high assurance data collected from endpoint systems. In our research we are building a prototype called the Enterprise Configuration Compliance Administration Manager (EC-CAM) system. Managed clients in the EC-CAM system utilize endpoint system architectures with advanced chipsets that provide hardware virtualization support to create a secure partition from which to monitor the end user's system externally. External monitoring of client configuration ensures that the compliance function sees a complete and accurate picture of end user's environment, and our system architecture makes it far harder to subvert the monitoring software.

Our research focuses on monitoring for compliance of endpoint client systems with an organization's configuration and security management policies. However, it should be noted that our system architecture for external monitoring is applicable for many other security monitoring functions, where the integrity of an agent monitoring an end system must be protected.

In the next section we describe the EC-CAM system and endpoint architecture, followed by a description of the analysis process and a final section with discussion and conclusions.

2. EC-CAM SYSTEM AND ENDPOINT ARCHITECTURE

The EC-CAM system consists of a set of servers to collect endpoint configuration data and to distribute configuration compliance policy and data. The endpoint configuration compliance policy consists of an enumeration of authorized

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CSIRW'09, April 13–15, Oak Ridge, Tennessee, USA
Copyright 2009 ACM 978-1-60558-098-2...\$5.00.

hardware, software, and other configuration settings for endpoint clients. The configuration compliance data is the means of describing those configuration components in sufficient detail so as to determine that components have not been tampered with or otherwise compromised; for software, this means “signatures” – for example, a hash of the software to recognize the complete, authorized image. The EC-CAM system also includes agent software to run on each managed endpoint within the enterprise. Each client endpoint is configured to report configuration compliance data to an enterprise server for analysis and to receive policy and whitelist “signatures” for compliant software.

Client endpoints running the EC-CAM agent software have a specific system architecture to support the external monitoring functionality via virtual machine introspection – inspecting a virtual machine from the outside for the purpose of analyzing the software running inside [2]. Our endpoints run on hardware platforms with virtualization support, such as Intel VT-x technology. Additionally, our system architecture takes advantage of the Trusted Platform Module (TPM) – a computer chip that can securely store platform measurements to help ensure a trustworthy boot and that the platform remains trustworthy [8]. The traditional user system – e.g., a Windows XP system with all of its applications, user data, etc. – becomes a single virtual machine partition within the new client endpoint architecture, as shown in Figure 1 below.

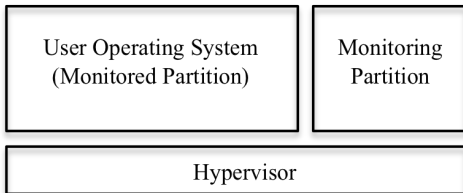


Figure 1. Endpoint Architecture

EC-CAM endpoints consist of a Xen 3.x hypervisor, a minimal installation of Linux in the monitoring partition, and Windows XP as the user operating system in the monitored partition.

The EC-CAM system monitors for typical elements of configuration compliance policies, such as authorized and unauthorized hardware, operating system, and applications, their versions, patches, and settings. The agent monitors the file system (on-disk inventory) for authorized and unauthorized software. The agent also monitors all the running software (including the operating system kernel, drivers, and user applications) to determine if that software is authorized or unauthorized. However, because the EC-CAM agent monitors a client externally, we can extend the configuration compliance function to verify not only that the running software is authorized, but also that the running software has not been tampered with.

Much of our compliance checking currently takes place off-line, using a copy of the memory image file, the swap file, and the virtual file system. While it is designed to work in the context of the specific architecture described here, large portions are general enough to be used in many other situations. All major Virtual Machine systems provide (1) the ability to capture a memory image file and (2) some mechanism for mounting and inspecting the file system. In addition, operating system support for hibernation provides the same sufficient level of access. As a result, our analysis process is just as applicable to after-the-fact

forensic analysis of systems that incorporate none of our architectural features as it is to this compliance checking application. The use of the swap file represents a useful extension to existing forensics memory analysis techniques [5].

3. ANALYSIS PROCESS

The architecture described above provides us with access to the raw data (memory and disk contents) in the monitored partition – the user’s operating system – with very high assurance. But this high assurance access represents a double-edged sword. While on the one hand, we do not have to depend on the (possibly unreliable) operating system APIs (application programming interfaces) to make sense of this raw data, it also means that we do not have access to these same APIs and must make sense of the raw data ourselves. Compliance policy is not stated in terms of raw data, but in much higher-level constructs. This difference is often referred to as the *semantic gap* [1].

In this section we describe some of the analysis processes that we use to extract higher-level constructs from the raw data without access to the operating system APIs. For Windows systems, they consist of the following:

1. Checking the file system of the monitored partition to ensure that required software is installed and not tampered with, and prohibited software is not present.
2. Checking the registry to ensure that key settings are in compliance with policy.
3. Checking the running software within the end user’s monitored system for authorized and unauthorized software. This involves analysis of the memory image (both captured from RAM and from the on-disk page file). Execution of unauthorized and unrecognized software is reported. Authorized software is checked to ensure that running program images have not been altered and that no unknown code has been inserted into the running system.
4. Checking and enforcing restrictions on access to certain hardware devices.

3.1 Checking Software on the File System

Checking the file system is fairly straightforward. The Xen hypervisor provides the ability to access the virtual disk partitions directly from the monitoring (host) partition. Mounting the user operating system’s virtual disk in read-only mode ensures that we will not cause any contention with the monitored user operating system in the guest partition. Once mounted, the monitoring partition has the ability to make sense of NTFS (Windows file system-format) disk structures.

While it is possible for the virtual file system to be in an inconsistent state, in practice this does not cause problems – Windows keeps its file system in a consistent state most of the time. Even when the file system has not been synced before pausing the user operating system, we have observed very few problems while walking an active file system (in read-only mode).

Once the virtual file system is mounted by the monitoring (host) operating system, it can be checked using traditional compliance approaches. Our prototype compliance system currently walks the file system looking for any Windows executable files (Portable Executable, or PE, format files). When located, these files are hashed and checked against a database of known

executables, both to ensure that required agents have not been tampered with and that prohibited software is not present.

3.2 Checking Software Settings

Windows keeps many key software settings in the system registry. Many existing policies specify both required and proscribed settings for specific keys. In order to enforce these, we need to be able to access the Windows registry.

Checking the registry via external monitoring is very complicated. Microsoft has not documented the format of registry hive files, and the vast majority of tools use the published APIs for any access to the registry. Nevertheless, there is some information available in the reverse-engineering community about how to manually parse registry hive files. Using this, we have been able to develop tools to extract specific keys and subtrees from the registry. This allows us to ensure that critical software is installed properly, and that specific configuration settings are set properly.

It is conceivable that direct access to registry hive files could fail to observe changes that were recently made but not yet committed to disk. In practice we have observed that Windows tends to commit registry changes to disk quickly. Furthermore, any pending changes would be committed before the next periodic check, at which time they would be reported.

A bigger concern is that the registry files might be in an inconsistent state and impossible to reliably parse. However, in practice this has yet to cause any significant challenges.

3.3 Checking Running Software

The greatest portion of our efforts have been focused on checking the executable code resident in memory. There are many, many techniques for obfuscating code on disk so as to evade compliance checking, but far fewer for memory. Our goal is to check that the in-memory images of executing processes (especially the kernel and other privileged code) have not been tampered with, and that other code in memory can be accounted for.

Our analysis takes advantage of the fact that major Windows XP data structures can be reliably located in memory. Some of these are prescribed by the physical hardware. Some are located through Windows' exported symbols. Others must be located by implicitly finding their address stored inside specific known kernel APIs. Together, they provide the starting point for understanding what processes, modules, and interrupt handlers are loaded and scheduled for possible execution. These represent the code base that we need to consider.

Using exported symbols, it is possible to locate the Windows Loaded Module List and Active Process List, which enumerate the device drivers and processes that are loaded and active. From these, we are able to access the saved registers, the page tables, the associated executable file, the loaded program image, and the list of dynamically linked libraries (DLLs). It is worth mentioning here that there is malware that is capable of hiding its presence from these data structures. But by locating and traversing the list of scheduled threads, any unlinked process list entries are readily apparent.

In order to check compliance, we examine the running memory images of processes to ensure that they have not been tampered with. To accomplish this, we must reconstruct the memory image of the process code (the .text segment) from the captured system memory image and the swapfile. Using the page tables associated

with the process, it is possible to locate each page in the process' virtual address space by either its location in physical memory (its offset in the system memory image) or its location in the swapfile.

The reconstructed .text segment is then compared to the original on-disk text segment. This is located on the disk image using the full path from the process table entry, and the .text segment extracted from that executable.

In theory, the .text segment extracted from the memory image should be identical to the .text segment extracted from the executable program file. In practice, there are a number of allowable differences that need to be considered:

- When pages from an executable program (not a device driver) need to be paged out, they might actually be *discarded*. The operating system knows that they have not changed and can be reloaded from the original executable. These holes in the memory space can be safely ignored, but they make comparison against an a priori computed hash of the entire .text segment impossible since we do not know which pages will be missing.
- Windows programs are not position-independent code. This causes problems for device drivers that are loaded into kernel memory and cannot predict their base address. Every device driver must be capable of being "rebased," providing a list of relocation offsets that need to be fixed up at load time. In order to ensure program image integrity, it is necessary to "unload" the program by reversing the fixup process. This is also true for dynamic link libraries (DLLs), which load into the address space of a process at a preferred base address but may suffer collisions with other DLLs that choose a conflicting base.
- Windows programs and libraries use Import Address Tables (IATs) to link against DLLs. The IAT is effectively a jump table that exists within the code (.text) of each application or library. When DLLs are loaded into memory, only the IAT needs to be changed with the actual addresses of each imported library function.

Each process' address space can include any number of dynamically linked libraries (DLLs), which also contain executable code that must be validated. By examining the module list associated with each process, we can locate each loaded DLL in the virtual address space. When Windows successfully loads a DLL at its preferred base address, it actually relies on a single physical instance of that code that is mapped into the process' virtual address space. We are able to take advantage of this and only check each physical DLL once. When DLLs collide and have to be rebased, Windows creates new physical pages that incorporate the rebased DLL in that process' address space. These copies are validated individually.

3.4 Checking Hardware

Finally, our system takes advantage of its position in the privileged partition to monitor and even enforce restrictions on allowable hardware devices. When, for example, USB devices are added to the system, the monitoring partition must decide whether to allow them to be made available to the monitored partition. Using the ivman [3] open-source handler for HAL (hardware abstraction layer) events, we are notified of these changes and given the opportunity to intervene.

We have used this capability to demonstrate policies that forbid the use of unauthorized portable USB drives and that monitor changes in the state of wireless networking. By leveraging the TPM, it is possible to demonstrate with very high reliability that certain restricted hardware devices (cameras, 802.11 wireless networking) can be disabled in software and made inaccessible to the user operating system in the guest partition.

4. RELATED WORK

Most related research addresses monitoring system memory for the purpose of rootkit detection or other security monitoring [2], [4], [6]. The work of Litty et al. [6] uses a hypervisor to intercept memory accesses and check individual pages for tampering. It “unloads” a page by reversing the effects of any memory relocation, and then computes a hash for the page, which is checked against a known good list. This is a promising approach for preventing malware from executing but like most prevention approaches it has the potential to affect the performance and resource constraints of the system. Our approach, by contrast, allows a “trust but verify” paradigm, where users can be given local administrative control over their systems but compliance with specific policies can be periodically verified.

Rutkowska’s System Virginty Verifier (SVV) [7] performs many of the same checks of system memory that we do. However SVV runs in the context of the operating system that is being verified, and depends on the visibility provided by the operating system, and can thus be undermined by rootkits.

5. DISCUSSION AND CONCLUSIONS

Although this is ongoing work, the results have been very promising. The semantic gap has proven challenging, but not insurmountable, and we have great visibility into the data structures that we need to monitor. It is simple to whitelist the few instances of “legitimate tampering” that we have observed. More significantly, we have found it quite easy to add detection of specific techniques that malware authors use to hide their activities.

There are a number of areas for discussion that would benefit from a workshop environment. They include:

- It is conceivable that a compromised system could in fact be spoofing all manners of Windows data structures, while maintaining real copies in some unnoticed corner of memory. However, we believe that such a compromised system would have to go to great lengths to avoid detection and maintain a consistent and comprehensive façade. In such cases, our architecture can take advantage of additional information (e.g. the values of all the privileged registers at the time the system snapshot was taken) that would allow additional anti-tampering checks.
- Thus far our work has focused on snapshots of paused systems. However, we have experimented with extraction of information from live operating system structures, such as being able to monitor process creation and destruction in real-time. While there are potential consistency issues and race conditions in running systems, we believe that most operating system structures are quite stable for the vast majority of the time. Real-time monitoring could take advantage of additional cores in modern CPUs, and could provide more constant compliance monitoring with virtually no performance impact.

- Our focus has been on executable code that might be inserted into legitimate application binaries. However, there are at least two additional cases that could circumvent this sort of analysis: interpreted languages and just-in-time (JIT) compilers. Interpreted languages store their program code as data. They provide a single legitimate executable that can be arbitrarily reprogrammed (within the constraints of the interpreted language) without having to alter their machine code at all. Our system, as designed, would have to use other levels of compliance checking to prevent or constrain interpreted programs. JIT compilers actually create new machine code on the fly. While we can observe the existence of this code, we cannot easily be sure that the code is a legitimately compiled version of an acceptable program, and not malicious code hiding in the data section of the JIT compiler.

We will investigate and address these issues in our ongoing research to provide higher assurance monitoring via endpoint architectures that provide separation of the monitored system from the monitoring agent functionality.

6. ACKNOWLEDGMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-08-9-0009. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

We would like to thank Brian Witten for his vision, articulation, and direction in this research project. We would also like to thank Denis Serenyi for his contributions in development of the TPM-related portions of our prototype.

7. REFERENCES

- [1] Chen, P. and B. Noble. “When virtual is better than real”. 8th Workshop on Hot Topics in Operating Systems, May 2001.
- [2] Garfinkel, T. and M. Rosenblum. “A virtual machine introspection based architecture for intrusion detection”. 10th Network and Distributed System Security, Feb. 2003.
- [3] Ivman. ivman.sourceforge.net.
- [4] Jiang, X., X. Wang, and D. Xu. “Stealthy malware detection through VMM-based “out-of-the-box” semantic view reconstruction”. 14th ACM Conference on Computer and Communications Security, Oct. 2007.
- [5] Kornblum, J. “Using every part of the buffalo in Windows memory analysis”. Digital Investigation Vol. 4-1, Mar. 2007.
- [6] Litty, L., A. Lagar-Cavilla, and D. Lie. “Hypervisor Support for Identifying Covertly Executing Binaries”. 17th USENIX Security Symposium, Jul. 2008.
- [7] Rutkowska, J. “System Virginty Verifier”. Hack In The Box Security Conference 2005 – Malaysia, Sep. 2005.
- [8] Trusted Computing Group. “Trusted Platform Module (TPM) Summary”, www.trustedcomputinggroup.org, 2008.