Protecting host-based intrusion detectors through virtual machines

M. Laureano, C. Maziero *, E. Jamhour

Graduate Program in Applied Computer Science, Pontifical Catholic University of Paraná, Brazil
Available online 16 October 2006

Abstract

Intrusion detection systems continuously watch the activity on a network or computer, looking for attack and intrusion evidences. However, host-based intrusion detectors are particularly vulnerable, as they can be disabled or tampered by successful intruders. This work proposes and implements an architecture model aimed to protect host-based intrusion detectors, through the application of the virtual machine concept. Virtual machine environments are becoming an interesting alternative for several computing systems due to their advantages in terms of cost and portability. The architecture proposed here makes use of the execution spaces separation provided by a virtual machine monitor, in order to separate the intrusion detection system from the system under monitoring. As a consequence, the intrusion detector becomes invisible and inaccessible to intruders. The prototype implementation and the tests performed show the viability of this solution.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Security; Intrusion detection; Virtual machines

1. Introduction

Several tools contribute to improve the security of a computing system. Among them, intrusion detection systems (IDS) stand out. Such systems continuously watch the system activity, looking for attacks and intrusion evidences. Network-based intrusion detectors scans data collected from the network to detect malicious activity, and can be installed on dedicated, well-protected machines. On the other hand, host-based intrusion detectors analyze local data collected from computing hosts. Because they run as processes in the monitored system, they are particularly vulnerable to successful intruders. Once an intruder enters the system, it is able to defeat or modify the intrusion detector in order to hide its presence.

Virtual machines can be used to improve the security of a computing system against attacks to its services [6]. The virtual machine concept was defined in the 1960s: in the IBM VM/370 environment, virtual machines were used to create an exclusive environment for each user [12]. The use of virtual machines is becoming interesting also in modern computing systems due to their advantages in terms of cost and portability [5]. Examples of currently used virtual machines environments are...
VMware [26] and UML – User-Mode Linux [7]. A frequent use of virtual machine-based systems is the so-called server consolidation: instead of using several physical equipments, one uses a single (and more powerful) hardware equipment, in which several distinct, isolated virtual machines host distinct operating systems, applications, and services.

This work proposes and implements an architecture model aimed to protect host-based intrusion detectors, through the application of the virtual machine concept. The architecture proposed here makes use of the execution spaces separation provided by a virtual machine monitor, in order to separate the intrusion detection system from the system under monitoring. This separation protects the intrusion detector, as it becomes invisible and inaccessible to guest processes (and to eventual intruders). Through modifications on the virtual machine monitor, it is possible to transparently collect detailed information about the guest operating system activity. This data is then sent to an external intrusion detector, running on the underlying host operating system. Comparing it against a previously generated behavior database (created from previous executions), the intrusion detector can look for behavior deviations in guest processes. If an intrusion is suspected, a response system can act in order to prevent or defeat it. This feature is easily implemented by intercepting the system calls issued by guest processes.

This article is structured as follows: Section 2 recalls some virtual machine concepts used in this work, Section 3 reviews intrusion detection techniques, Section 4 details the proposal, Section 5 describes the current implementation, Section 6 presents experimental results, and Section 7 discusses related works.

2. Virtual machines

A virtual machine (VM) is defined in Popek and Goldberg [20] as an efficient and isolated duplicate of a real machine. Typical uses for virtual machine systems include the development and testing of new operating systems, simultaneously running distinct operating systems on the same hardware, and server consolidation [24].

A virtual machine environment is created by a Virtual Machine Monitor (VM Monitor or VMM), also called an “operating system for operating systems” [14]. The monitor creates one or more virtual machines on a single real machine. Each virtual machine provides facilities for an application or a “guest system” that believes to be executing on a standard hardware environment. VM monitors build some properties that are useful in system security, like isolation (a software running in a VM cannot access or modify the monitor or other VM), inspection (the monitor can access the entire VM state), and interposition (the monitor can intercept and modify operations issued by a VM) [14,11].

There are two standard approaches to organize virtual machine systems: type I, in which the virtual machine monitor is implemented between the hardware and the guest system(s), and type II, in which the monitor is implemented as an ordinary process of an underlying real operating system, called the host system [6]. A more detailed classification of virtual machine systems is presented in Smith and Nair [23].

Standard PC processors provide no adequate support for virtualization [21]. Consequently, virtualization overhead can be as high as 50% of total computing time [5,7,26]. However, recent research significantly reduced such costs, as shown King et al. in [15,16], and Whitaker et al. in [28,27]. Through improvements in the binary interface between the monitor and the guest operating systems, the Xen Project [2] obtained average computing costs below 3% for virtualizing Linux, FreeBSD, and Windows XP. These works open many perspectives on the use of virtual machines in production environments, as discussed by Rosenblum [22].

3. Intrusion detection

An Intrusion Detection System (IDS) continuously collects and analyzes data from a computing system, aiming to detect intrusive actions. With respect to the origin of analyzed data, there are two main approaches for intrusion detection [1]: network-based IDS (NIDS), which are based on watching the network traffic flowing through the systems to monitor, and host-based IDS (HIDS), which are based on watching local activity on a host, like processes, network connections, system calls, log files, etc. The main weakness of host-based intrusion detection is its relative fragility: in order to collect system activity data, the HIDS software (or an agent on its behalf) should be installed in each machine to monitor. This agent can be deactivated or tampered by a successful intruder in order to mask her presence.
Techniques used to analyze collected data in order to detect attacks and intrusions can be roughly classified in: signature detection, when collected data are compared to a base of known attack patterns (or signatures), and anomaly detection, when collected data are compared to (previously stored) data representing the normal activity of the system. Deviations from normality are then signaled as threats.

4. Protecting intrusion detectors through virtual machines

As previously shown, host-based IDSs are vulnerable to local attacks, since the intruder can disable or tamper them. The use of virtual machines provides a solution to this problem. The proposal presented here allows building more reliable host-based intrusion detection systems.

The proposal’s main idea is to encapsulate the system to monitor inside a virtual machine, which is monitored from outside (the host system). The intrusion detection and response mechanisms are implemented outside the virtual machine, i.e. out of reach of intruders. This proposal adopts a type II virtual machine monitor, running on top of a host operating system. This approach turns easier the implementation, because the detection and response system can be implemented as ordinary processes on the host system user space. Also, facilities available on the host system, like IPC mechanisms, packet filters, logging services, and databases, can be easily integrated to the architecture. Fig. 1 illustrates the main components of the proposed architecture.

The interaction of guest system processes with the outside world is done only through the network, using a software firewall managed by the host kernel (like Linux IPTables, for instance [18]). Under the guest system’s viewpoint, it is an external firewall, therefore inaccessible to intruders.

The main architecture modules are the intrusion detector module, which compares data collected from the guest system against a previously stored knowledge database, the access control module, which checks if processes and users are known and respect a previously built access control list, and the response module, which receives alarms issued by the intrusion detector or the access control module and transforms them in actions on the guest system and/or the host firewall.

The interactions between the guest system and the intrusion detection and response modules are carried out through the virtual machine monitor. Two types of interaction are defined: monitoring, in which guest data is supplied by the virtual machine monitor for external analysis and storing,
and response, as the response module can act on the
guest system in response to intrusions. Beyond
actions on the guest system, the response module
can also interact with the host firewall, blocking
ports and connections to the guest system as needed.

4.1. Detecting intrusions on the guest system

The proposed architecture does not impose a spe-
cific algorithm or approach for intrusion detection.
For instance, guest processes behavior data can be
extracted from the guest system and compared to
behavior data previously stored in the knowledge
base, in order to detect anomalies. Alternatively,
the knowledge base can store descriptions of actions
known to be harmful (i.e. attack signatures), which
can be compared to the actions being performed by
the guest processes. In order to show the feasibility
of our proposal, a well-known anomaly-based
approach intrusion detection was chosen, which will
be described in the following.

The system calls issued by a process constitute a
rich source of information about its activity. Several
papers describe techniques for anomaly-based intru-
sion detection using such data. In Forrest et al. and
Hofmeyr et al. [10,13], the authors propose the fol-
lowing algorithm: all system calls issued by a pro-
cess are sequentially logged, discarding their
parameters. This execution history is then “sliced”
into sets of sequences of syscalls with length \( k \).
The collection of all possible sequences with length
\( k \) defines the normal behavior for that process.
Any sequence of \( k \) system calls issued by that pro-
cess and not present in its normal behavior (previ-
ously stored sequences) is considered as an
anomaly. To illustrate that technique, let us con-
sider a UNIX process which issued the following
system calls during its execution:

\[
\text{open read mmap mmap open read mmap}
\]

Adopting \( k = 3 \), the following set of sequences of
syscalls is obtained:

\[
\text{[open read mmap mmap open read mmap]}
\]

If the process issues a different sequence, like
[open open read], it should be placed under suspi-
cion. Despite the set of system calls to be system-
dependant and the capture of the complete behavior
of a process to be potentially laborious, this method
showed to be efficient, as shown by their authors. A
thorough analysis of this algorithm can be found in
Forrest et al. [10]. Nevertheless, it should be empha-
 sized that, although our current implementation
adopts the anomaly-based intrusion detection stra-
 tegy described here, the architecture presented in
Fig. 1 is generic enough to easily accept other com-
 mon approaches.

4.2. Access control

Beyond anomaly-based intrusion detection, guest
data provided by the virtual machine monitor can
be used to carry out other analysis. One interesting
possibility is to compare guest system activity
against a previously stored access-control list
(ACL) which defines which users are allowed to
run which executables. Users and/or executables
not in the ACL should have their processes labeled
as suspect. This facility is provided by the access
control module in our architecture. However, as
the architecture does not impose any specific access
control model, more complex models and policies
can be adopted as well.

4.3. Learning and monitoring

Our system has two operation modes: a learning
mode and a monitoring mode. When in the learning
mode, the system stores the sequences of system
calls for guest processes in the knowledge database.
Also, all the processes executing in the guest system
and their respective users are recorded as authorized
processes and users, thus automatically generating
an access-control list (ACL). Therefore, the learning
mode allows recording the “normal behavior” of
the system, collecting data for further intrusion
and ACL violation detections. Obviously, only
ACL data needs to be stored if the system adopts
a signature-based intrusion detection strategy.

When in monitoring mode, the intrusion detec-
tion module receives data from the virtual machine
monitor and compares it to the “normal” data pre-
viously stored in the knowledge base during the
learning phase. The current prototype analyzes
sequences of system calls issued by guest processes,
using the algorithm presented in Forrest et al. [10].
If a system call sequence issued by a given process
is not found in the knowledge base, an anomalous
situation is signaled and that process is declared suspect. Also, processes not respecting the previously generated ACL are declared suspect by the access control module.

4.4. Restricting processes under suspicion

Suspect processes are to be restricted in their access to the guest system, to prevent harmful actions. Such restriction is currently implemented by denying suspect processes access to some system calls. In the papers [3,4], Bernaschi classifies the UNIX system calls in functionality groups (communication, file system and memory management are some of such groups) and levels of threat. In their proposed scale, system calls classified in threat level 1 can be used to get privileged access to the operating system; the level 2 contains system calls that can be used for denial of service attacks; system calls able to compromise the caller process are classed in threat level 3; finally, system calls in level 4 are harmless for system security.

The Bernaschi’s classification is being used here as follows: all the system calls classified as threat level 1 (shown in Table 1) are denied for suspect processes: all invocations to such system calls issued by processes labeled as suspect will return a ‘permission denied’ or equivalent error status; such event can also be logged for further analysis. Since only system calls on threat level 1 may be useful for gaining privileged access to the guest operating system, only such syscalls are denied to suspect guest processes. This mechanism is implemented by the virtual machine monitor, through the interception of the system calls issued by guest processes. Using this approach, the guest operating system can isolate a suspect process without causing severe impact on other guest processes.

The architecture presented here keeps the detection and response system out of reach of intruders. However, to guarantee the system security it is important to observe that interactions between the guest and host systems must be carried out only through the virtual machine monitor. Also, the virtual machine monitor, the host system, and other VMs running on the same host must remain inaccessible to guest processes (according to the isolation property of virtual machine monitors). Finally, all network services must be provided only by guest system processes; the network access to the underlying host system should be carefully controlled.

5. Current implementation

A prototype was implemented in a Linux platform, using the User-Mode Linux virtual-machine monitor (UML) [7]. UML implements a type II monitor, which allows running Linux guest systems on top of a Linux host. It should be noticed that UML performance is worse than commercial products like VMWare [26], but it is open source. UML source code was modified to allow extracting detailed data from the guest system, like the system calls issued by each guest process. Communication between the UML monitor and the external processes running on the host system was done through named pipes (this way, the host operating system synchronizes the data flow between them).

Two different implementations were built: a synchronous and an asynchronous one. In the synchronous implementation, each system call issued by a guest process is sent by the monitor to the external IDS; the guest process pauses until the system call is validated. This approach is simpler to implement, but imposes a high performance cost on guest processes. On the other hand, the asynchronous implementation is more complex but offers better performance. In such approach, the monitor sends each system call issued by guest processes to the external IDS; guest processes are not imposed to wait for system call validations. If the IDS detects suspect actions coming from a guest process, it will warn the monitor through an UNIX signal. This approach leads to a small time gap between a

<table>
<thead>
<tr>
<th>Group</th>
<th>System calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>File system and devices</td>
<td>open link unlink chmod lchown rename fchown chmod mknod mount symlink fchmod</td>
</tr>
<tr>
<td>Process management</td>
<td>execve setgid setreuid setregid setgroups setfsuid setfsgid setresuid setresgid setuid</td>
</tr>
<tr>
<td>Module management</td>
<td>init_module</td>
</tr>
</tbody>
</table>
(possible) malicious action performed by a guest process and its countermeasures (classification of such process as suspect).

The current ACL implementation consists simply on a table containing pairs [uid, path] of authorized users and executable file paths, supporting wildcards on both fields. Any process not matching an ACL entry will be labeled as suspect. This ACL can be easily extended to include other guest resources, like network port numbers, and so on.

6. Experimental results

Using the prototype, a time measurement was carried out regarding the execution of basic user commands, in order to evaluate the performance impact of the proposal. The utilities ps, find, ls, and who were selected because they are UNIX tools frequently tampered by intruder rootkits, and because they can generate a large number of system calls during their execution.

The command execution times were measured in five situations: (a) in the host system, (b) in the original guest system, (c) in the guest system on learning mode, (d) in the guest system on monitoring mode, and (e) in the guest system on monitoring mode, but using an asynchronous implementation. Observed variances were under 5% in all time measurement. The hardware used in the experiments was a dual-processor server (Dual P3 1130 MHZ, 2 GBYTES RAM). The host system was running a 2.6.9 SMP Linux kernel, and the guest systems used single-CPU 2.6.9 Linux kernels.

Table 2 presents the average execution times for each command and their relative overheads. The number of syscalls issued by each command execution is also presented. Execution times observed in the guest system (b) are far superior to those observed in the host system (a); this is due to the high virtualization overhead presented by UML. Also, for the synchronous implementation, the overheads imposed by modifications in the virtual machine monitor to interact with the external learning, detection, and response mechanisms are quite high, in both modes (c and d). This cost is due to the non-optimized implementation of the learning and monitoring routines and of their interaction with the UML monitor.

In order to evaluate the impact of our proposal on guest processes using the network, some tests were carried out using the wget tool (a command-line HTTP/FTP client). The tests consisted on downloading 100 Kb and 1 Mb remote files. Table 3 summarizes the results, which show overheads below 10% when using the asynchronous implementation. Additionally, in order to evaluate the effectiveness of the IDS in detecting and defeating intrusions, some tests have been carried out using some popular rootkits (described in Table 4 and available at http://www.antiserver.it/Backdoor-Rootkit/).

These rootkits modify commands of the original operating system to prevent their detection (hiding

<table>
<thead>
<tr>
<th>Command</th>
<th>ps –ef</th>
<th>find &gt;/dev/null</th>
<th>ls -laR &gt;/dev/null</th>
<th>who –t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of system calls</td>
<td>536</td>
<td>10055</td>
<td>17225</td>
<td>96</td>
</tr>
<tr>
<td>(a) Host</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>25</td>
<td>125</td>
<td>802</td>
<td>5</td>
</tr>
<tr>
<td>Time</td>
<td>68</td>
<td>484</td>
<td>1160</td>
<td>29</td>
</tr>
<tr>
<td>Overhead relative to (a)</td>
<td>172%</td>
<td>287%</td>
<td>44%</td>
<td>480%</td>
</tr>
<tr>
<td>(c) Learning mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>81</td>
<td>812</td>
<td>1784</td>
<td>32</td>
</tr>
<tr>
<td>Overhead relative to (b)</td>
<td>19%</td>
<td>67%</td>
<td>53%</td>
<td>10%</td>
</tr>
<tr>
<td>(d) Synchronous monitoring mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>107</td>
<td>857</td>
<td>1790</td>
<td>33</td>
</tr>
<tr>
<td>Overhead relative to (b)</td>
<td>57%</td>
<td>77%</td>
<td>54%</td>
<td>13%</td>
</tr>
<tr>
<td>(e) Asynchronous monitoring mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>68</td>
<td>532</td>
<td>1232</td>
<td>30</td>
</tr>
<tr>
<td>Overhead relative to (b)</td>
<td>0%</td>
<td>10%</td>
<td>6%</td>
<td>3%</td>
</tr>
</tbody>
</table>
the intruder’s processes, files, network connections and so) and to steal typed information like logins and passwords (through modifications in commands like telnet, sshd and login). All tools available in those rootkits were executed with standard parameters, and all the modifications inserted by them were detected in all the executions.

The tests evidenced the effectiveness and complementariness of both mechanisms implemented in the system: the intrusion detection mechanism detects and hinders the execution of known but tampered binary files, while the access control hinders the execution of unknown binary files, or processes launched by unknown or unauthorized users.

7. Related works

The paper [6] cited some benefits the use of virtual machines can bring to the security and compatibility of systems, as the capture and processing of log messages, intrusion detection (through the control of the virtual machine’s internal state) or system migration easiness. However, the article does not demonstrate how these proposals should be structured and implemented, nor analyzes their impact on system performance.

The reference [8] describes an experience of use of virtual machines for the security of systems. The proposal defines an intermediate layer between the monitor and the host system, called Revirt. This layer captures the data sent through the syslog process (the standard UNIX logging daemon) of the virtual machine and sends it to the host system for storing and later analysis. However, if the virtual system is compromised, the guest syslog daemon can be terminated and/or the log messages can be manipulated by the intruder, and consequently they are no longer reliable.

The work described in Garfinkel and Rosenblum [11] is the closest to our approach. It defines an
architecture for intrusion detection in virtual machines called VMI-IDS (Virtual Machine Intrusion Detection System). Their approach considers the use of a type I monitor, executing directly on top of the hardware. The IDS executes in a privileged virtual machine and scans data extracted from the other VMs, searching for intrusion evidences. Only the low-level internal state of each virtual machine is analyzed, without taking into account the activities carried out by their guest processes. Also, the system response ability is limited: in case of intrusion suspicion, the suspect virtual machine is suspended for deeper analysis; if the intrusion is confirmed, the entire virtual machine is restarted from a safe state.

That approach differs from our proposal in several aspects, like the nature of collected data, the intrusion detection methods, the access control feature, and more specific intrusion response. Our proposal allows analyzing processes separately, detecting anomalous activities and hindering intrusions from compromised processes. This way, disturbance on unrelated guest processes is minimized. Moreover, there is no need to suspend the entire virtual machine for intrusion confirmation. Another unique feature in our proposal is the use of an authorization model (ACL) for users and processes, automatically generated during the learning phase.

An alternative approach to protect intrusion detectors from local attacks could be carried out through the use of multiple user-contexts. Some recent operating system kernels [19,9,17,25] can define several autonomous and isolated user contexts. In such approach, the intrusion detector and the response system would be installed on a more privileged context, from which they could monitor and act on processes running in the other contexts. This approach can achieve good performance results, but imposes the same operating system kernel to all user contexts.

8. Conclusion

This paper describes a proposal to increase the security of computing systems using virtual machines. The basis of the proposal is to monitor guest processes’ actions through an intrusion detection system, external to the virtual machine. The data used in intrusion detection is obtained from the virtual machine monitor and analyzed by an IDS process in the underlying real machine. The detection system is inaccessible to virtual machine processes and cannot be subverted by intruders. Also, the intrusion detection module is able to track the activity of isolated processes, and the response module can restrict their execution without disturbing other non-related guest processes.

The main objective of the project, to hinder the execution of suspect process in the virtual machine and consequently avoid the system compromise, was reached with the current prototype. However, complementary work must be done to improve the performance of the current intrusion detection and response mechanism and thus to minimize its overhead. Another aspect to be refined is to define more flexible ways to interact with the guest kernel, allowing killing or suspending specific guest processes. Also, the interactions between the response module and the host system firewall, to block suspect network traffic, need to be detailed and implemented.

Other questions to be studied include implementing detection mechanisms based on other relevant data, like the network traffic generated by the virtual machine, and the behavior of guest users. Faster and more sophisticated algorithms for intrusion detection could be implemented based on such information, helping to reduce the occurrence of false results (both positive and negative). Finally, in order to ease the use of the system, the next prototype should allow both monitoring and learning modes to occur simultaneously, for distinct processes. This would allow the system to “learn” about a recently installed application, while monitoring the other guest processes.

References


Marcos Laureano received a M.Sc. degree on Computer Science from the Pontificial Catholic University of Paraná State, Brazil, in 2004. He gives lectures in Operating Systems, Computer Security, and Programming Languages in local universities, and works in the Information Technology field since 1991. He published technical books on virtual machines and C programming.

Carlos Maziero is a full professor of Operating Systems at the Graduate Program in Computer Science of the Pontificial Catholic University of Paraná State, Brazil, since 1998. He received a doctorate degree in Computer Science from the Université de Rennes I (IRISA/INRIA) – France, in 1994, and a M.Sc. degree in Electrical Engineering from the Santa Catarina Federal University – Brazil, in 1988. His research interests include security of network operating systems and distributed systems, resource management in mult-user operating systems, and operating systems teaching techniques.

Edgard Jamhour received a Ph.D. degree in Informatics and Automation from L’Université de Franche Comté – France in 1994. He is currently a full professor of the Graduate Program in Computer Science at the Pontificial Catholic University of Paraná State, Brazil. His research interests include network protocols, management and security.