A security extension to the Linux Kernel Virtual Machine

Flavio Lombardi
CNR - Consiglio Nazionale delle Ricerche
Piazzale Aldo Moro 7
Roma, Italy
flavio.lombardi@cnr.it

Roberto Di Pietro*
UNESCO Chair in Data Privacy
Universitat Rovira i Virgili
Tarragona, Spain
roberto.dipietro@urv.cat

Francesco Cipollone
Technet srl
Roma, Italy
francesco.cipollone@techneteu.com

ABSTRACT

Virtualization is becoming widely used in regular desktop PCs, data centers and server farms, where the additional layers introduced by virtualization can potentially increase overall system security. In this paper, we propose KVM-SEC, an extension to the Linux Kernel Virtual Machine that is focused on increasing the security of the Virtual Machine. KVMSEC main features are : it is transparent to guest machines; it is hard to compromise even from a malicious virtual machine; it can collect data on guest machine and analyze it; it can provide secure two-way communication between the host and the guest; it can be deployed on Linux supported host machines and supports Linux as guest machine. These features are leveraged to implement a real-time monitoring and security management system. KVMSEC features, architecture, and working prototype are shown in detail in this paper; differences and advantages over previous solutions are highlighted as well. Finally, note that the results exposed in this paper are open source and available on SourceForge.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems —*Security and Protection*; K.6.5 [**Security and Protection**]: Unauthorized access —*hacking*

General Terms

Virtual Machine

Keywords

KVM, Xen, hypervisor, virtualization, security

1. INTRODUCTION

Virtualization is an old idea that is living a new golden era. It is becoming widely used in regular desktop PCs, data centers and server farms. Furthermore, the trend for this technology predicts a future of strong growth [11]. Such a success is mostly due to the reduced total cost of ownership and to the ease of management of the virtual machines with respect to their physical counterparts [6]. The most widely adopted virtualization solutions are targeted to the x86 platform: Xen [4], Virtuozzo [21], UMLinux [9], Qemu [5], VMware [7], and KVM [16] to cite a few. Most solutions are free open source products and share the same underlying technology with their commercially supported counterparts, with the notable exception of some VMware products. Even though the reliability of virtualization solutions has increased in recent years, such technologies still pose many new challenges, especially when assessing security. In particular, the level of robustness to attacks compromising the security of services and operating systems inside virtual machines is a vexed issue. However, note that the additional layers introduced by the different virtualization architectures can potentially increase the overall system security because a virtual machine (also VM or guest in the following) can be protected by additional boundaries [3]. In particular, an hypervisor (also Virtual Machine Monitor or VMM in the following) could provide the equivalent of a trusted computing base.

Security is always at risk and increasingly often malicious intruders succeed in attacking a system and gaining administrator privileges. The intruder can then violate the integrity of the filesystem [20] and alter one or more OS core utility or user program (e.g. replacing regular system utilities like *ps,ls* with malicious insecure counterparts). In this way, the attacker can insert trojans, backdoors, alter important file content or access classified or private information. Since a modification of a system utility implies changes to some part of the attacked filesystem, almost all security tools inspect modification of files in critical paths (e.g.. */bin* in Linux or *C:Windows* in Operating Systems from Redmond) in order to detect a malicious alteration. Usually the Integrity Tool and data it collects reside on the same machine that is being monitored. This can be a problem in case such machine is compromised. Migrating the monitoring system outside of the monitored system (e.g. by exploiting virtualization) may help, so even if an attacker succeeds in compromising a VM:

- it cannot delete the traces (e.g.. log files) of its successful attempt (since data is stored on the host side);
- it cannot deactivate the external security system (since it is on the host side).

*Also with Dipartimento di Matematica, Università di Roma Tre. E-mail: dipietro@urv.cat

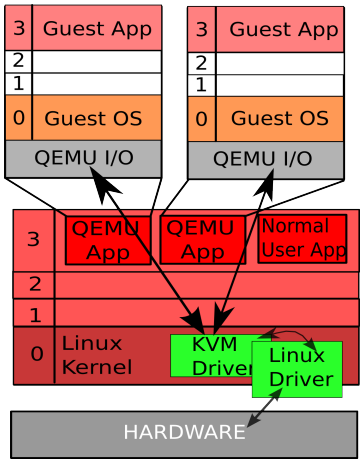

Figure 1: KVM Execution Environment

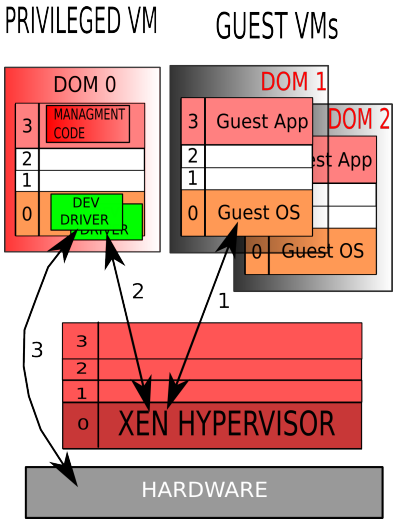

Figure 2: Xen Execution Environment

In this paper, we propose a novel approach to Real-Time Integrity Monitoring that allows the host to control unauthorized changes to guest virtual machines. Our system, named KVMSEC, can detect anomalies in order to correct them or just notify about their occurrence. KVMSEC is an extension to the Linux Kernel Virtual Machine system aiming at being as less visible and vulnerable as possible. It is composed of multiple modules that are executed inside both the host and guest kernels. By communicating through secure channels, such modules are able to provide the host with up to date and accurate information about the guest. The core modules of the detection system are located on the host machine (so that an attacker confined in a VM can hardly reach them) whereas data is collected by modules located at both guest kernel and userspace level. KVMSEC main features are: it is transparent to guest machines; it is hard to compromise even from a malicious virtual machine; it can collect data on guest machine and analyze it; it can provide secure two-way host-guest communication; it can be deployed on both x86 and x86_64 machines (depends on Linux support); it supports Linux as guest and host OS. It is worth noting that in KVMSEC each Virtual Machine uses its own private memory area to communicate with the host, so it is totally independent from other VMs. A module in guest userspace is devoted to collecting data of interest. The fact that such a module runs at userspace level can help reducing the computational load at the kernel side. However, notice that there can be a tradeoff between monitoring system stealthiness and reduced load on guest kernel, since a userspace process can be less difficult to spot and tamper with than a daemon in kernel space. So, if needed, the data collection task can be carried out by the KVMSEC kernel module itself.
Finally, KVMSEC source code is open and freely available to the community working on virtualization —and in particular on security issues in virtualization— on SourceForge.
The rest of the paper is organized as follows: Section 2 surveys present technologies and related work and provides background information for our work; Section 3 describes the requirements and architecture of the KVMSEC project; Section 4 provides some implementation details, such as the main data flows and protocols used; Section 5 discusses KVMSEC features compared with previous results; Finally, in Section 6 conclusions are drawn.

2. BACKGROUND

As mentioned above many virtualization architectures are available today; we focused on solutions whose source code was freely available for us to experiment with.
In the following, we analyze the most important open source virtualization architectures, i.e. KVM and Xen. By showing the main characteristics of the two systems we also justify why we chose the former (innovative) system over the latter (and well established) one. In the following *full-virtualization* is a technology that exploits CPU hardware support to virtualization (AMD-V and Intel-VT)[2]. As a consequence the guest OS can potentially run unmodified on a VM. The majority of modern CPUs support such technology featuring a new execution level that helps trapping I/O requests. Operating Systems in guest VMs are then executed at the original CPU ring level, and are unaware of the existence of an additional privileged level. The guest OS runs then unmodified on the KVM guests. Para virtualization usually requires changes to operating systems in order for them to run inside guest machines

2.1 Virtualization Architectures

KVM stands for Kernel Virtual Machine, it is the Linux officially supported virtualization solution being part of the official Linux kernel source since version 2.6.20. KVM makes use of the standard Linux drivers to manage hardware resources and supports the extended virtualization cpu instruction set described above. KVM consists of:

1. The KVM hypervisor which is a Linux kernel module;
2. Qemu-KVM: a modified version of the Qemu [5] emulation software.

KVM works both in *full-virtualization* and *para-virtualization* mode and drivers and interface with the HW are delegated to the Linux OS, guaranteeing active development and support from the large kernel user base. KVM introduces another execution level, Guest mode (see Figure 1). The main modes of operation follow:

1. Guest mode: running code that does not imply I/O operation;

2. Kernel mode: changes the execution context (from guest to host) when the guest performs an I/O operation;

3. User mode: performs the I/O operation on behalf of the guest OS.

The Xen virtualization system can work either in *full-virtualization* or *para-virtualization* mode (see Figure 2). In the latter mode the guest OS must be modified with ad-hoc drivers that will invoke the Xen hypervisor when needed (e.g.. when the guest OS performs an I/O operation). Xen is composed of three modules:

1. The Xen Hypervisor (or VMM) that provides controlled resource access to virtual machines;

2. Dom0 that is a privileged VM;

3. DomU that is a common VM where guest OS is executed.

Xen hypervisor features can be summarized as follows:

- Shared memory: a communication channel between VMs;
- Ether channel: a signaling channel between VMs;
- Shared Memory Access control: in the hypervisor there is an access matrix that describes which VM can access the shared memory, how much space is granted and what transactions are permitted.

Since KVM is a kernel module, drivers are those contained in the Linux kernel so they are always updated to the latest version. On the one hand, this is one of the reasons why KVM is supposed to be less vulnerable to attacks than Xen, whose driver development is slower than in standard Linux. On the other hand, the kernel in it's entirety is more complex compared to the Xen hypervisor and as a consequence can be more subject to vulnerabilities and bugs. Xen drivers are confined in the privileged Dom0 domain and separated from the others. All I/O requests by the various domains are performed by Dom0 and the hypervisor has the responsibility to switch from a DomU to Dom0 when a I/O operation is requested.

Main pros and cons of these two virtualization architectures are summarized in Table 2.1 where it is clear that KVM, despite being not fully mature yet, offers advantages over Xen.

2.2 Related Work

In the following we briefly survey some interesting solutions to the problem of system integrity monitoring, that can be reconducted to the broader area of Intrusion Detection. Filesystem Integrity Tools exist today like Tripwire [14] that can monitor changes in the critical path by computing file checksums at regular intervals . The Advanced Intrusion Detection Environment (AIDE)[22] is a local IDS for Linux that creates a database of checksums about selected files according to the chosen policy. Any changes not allowed by the configuration file are reported. Osiris [23] is a host integrity monitoring system that can centrally manage a number of remote clients over secure but visible network connections (ssl). Samhain [24] centralized Filesystem Integrity and intrusion detection supports both single local and multiple remote machines, using checksumming and an interesting stealth mode aimed at protecting remote clients on remote machines. Finally, for an updated vision on the state of the art and research problems in the Intrusion Detection System domain, refer to [8].

Feature	**Kvm**	**Xen**
Driver Availability	standard Linux kernel	derived from Linux kernel
Upgradeability	modules allow easy possibly real-time upgrade	requires restarting host
HW Support	any supported by standard kernel	most of the standard kernel drivers
CPU Support	*full-virt* requires recent CPU	*para-virt* mode works on all x86 CPUs
Flexibility	seamlessly hosted on 2.6.x kernels	already packaged in most distributions
Maturity	recently inserted into streamline kernel	already a mature commercially-supported project

Table 1: KVM vs Xen

Various proposals leverage Xen hypervisor isolation capabilities: The sHype [13] system for Xen offers the isolation of VMs with MAC enforcement and proposes an approach to manage covert channels whereas Yang [25] modifies Xen to protect user application data privacy by removing the operating system from the trust base. Quynh and Takefuji propose a real-time Filesystem Integrity Tool (FIT in the following) named XenFIT [18] or XenRIM [17] protecting the database and the FIT system from the attacker by exploiting the isolation given by the Xen hypervisor. In fact, FIT and database are deployed onto a separate VM. XenRIM is composed of a Linux Security Module (XenRIMU) in DomU, collecting information about the client, and a daemon process (Xenrimd) in Dom0 that checks the information collected by XenRIMU and reports any violations of the security policy. XenRIM main features are:

1. Real Time detection: XenRim is implemented as kernel module which allows defining hooks on OS functions;

2. Low impact on performance: XenRim intercepts only hooks related to accesses to the critical path;

3. Resistance to tampering: core XenRim modules work

in a separate Xen domain, isolated from other VMs, data collectors are the only part of XenRim located on the monitored VM;

4. Stealthiness: it is difficult to discover the presence of a monitoring system given the lack of a userspace guest daemon process;

5. Communication: global shared memory and event channels are set up between Dom0 and the various DomU*s*.

In 2007 Quynh and Takefuji developed XenKimono [19], based on ideas from Garfinkel and Rosenblum [10]. This software is an IDS aimed at discovering malicious intrusions by analyzing the VM kernel internal data structures from outside. All XenKimono modules are inside the host machine and analyze the VM raw memory for malicious software (e.g.. rootkits). The translation of raw VM memory in higher level OS kernel compatible data structures is done by extracting kernel symbols from the DomU kernel binaries and exploiting the LKCD project library [1]. This way XenKimono is able to locate DomU kernel data structures in raw memory. One of the main motivations behind our work is the need for improving the security level of the increasingly deployed virtual machines, and in particular those supported and supporting Linux as both a guest and a host. Striving towards this objective we designed and developed the KVMSEC system, whose architecture is described in the following section.

3. KVMSEC ARCHITECTURE

In the following, we describe the main requirements of the KVMSEC project and some caveats over the problems we encountered and the way we solved them.

3.1 Requirements

Starting from previous results and experiences matured on the Xen hypervisor, we designed and developed KVMSEC, a system extending the Linux Kernel Virtual Machine and aimed at increasing the security of guest VMs. KVMSEC leverages core concepts of Intrusion Detection Systems [8] and is at the heart of a larger work that will yield a complete protection system for virtual machines.
KVMSEC was designed to match the following requirements:

R1 Transparency: the system should be as less visible and accessible as possible from the VM viewpoint, the potential intruder should neither detect any monitoring system nor being able to access it;

R2 Immunity to attacks from the Guest: in case an attacker succeeds and compromises one of the guests, the host system should not be exposed to attacks and should keep working as usual;

R3 Ease of Deployment: KVMSEC should be easily deployable and should work on a vast majority of available hardware;

R4 Dynamic reaction: When an intrusion is underway inside a VM, KVMSEC should limit the damage and should notify or take action against the compromised guest.

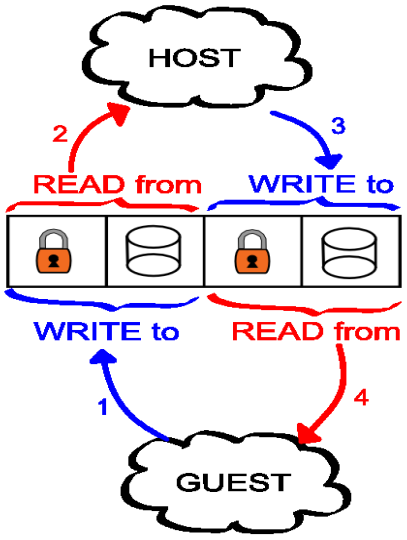

Figure 3: KVMSEC Shared Memory Access

3.2 Caveats

During the analysis of KVM internals we found that, unlike Xen, there is no KVM functionality that supports shared memory between VMs. Furthermore, KVM does not have any tool similar to the Xen *ether channel* that could carry signals among the various VMs. In the following we list and briefly explain these and other caveats, as well as the solutions we found for them and adopted in KVMSEC.

PS1 Lack of shared memory support in KVM between guest VMs and Host lead us to design a communication system based on shared memory between host and guest. We did not implement such a communication system as the *event channel* for Xen —also because the implementation of a signaling channel would have made the whole KVMSEC more visible to an attacker in the guest (against R1).

PS2 Lack of a signaling channel between guest VM and Host in KVM lead us to the choice of letting KVMSEC poll the shared memory at regular intervals to check for new messages.

PS3 Lack of access control to shared memory in KVM lead us to synchronize host-guest accesses to shared memory. In order to simplify access control management, each KVMSEC VM has its own shared memory area for communications with the host OS. Furthermore, a simple lock mechanism is implemented for each of the two unidirectional channels in order to synchronize access to the shared memory area where messages transit.

In KVM, unlike Xen, the shared memory is not directly managed by the hypervisor but by the main emulation process i.e. Qemu-KVM. The choice of a communication channel using shared memory was made in order to comply with R1. In fact, using a virtual socket between host and guest would have resulted in a visible and attackable communication channel, (as happens in AIDE[22]). In addition, the

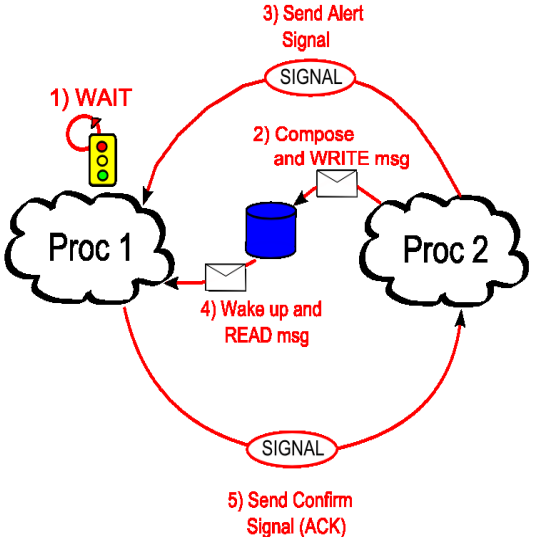

Figure 4: KVMSEC Communication Protocol

message handler is contained in a guest kernel module to make it as much secure as possible, in accordance with R2. At the host side the message handler is implemented in the shared memory management module within Qemu-KVM. The KVM shared memory (see Figure 3)is composed of four components:

- Two locks to protect the critical section;
- Two data buffers.

In order to meet requirement R4 i.e. being able to react to attacks, KVMSEC should be able to monitor critical processes running inside guest VMs. At present such functionality is not fully implemented, however KVMSEC is expected to be able to periodically check the existence and liveness of a number of daemon processes inside guests. If one of these processes is (abnormally) terminated, appropriate countermeasures are to be taken by the host, including collecting additional data for forensic analysis and, as a last resort, a Virtual Machine status freeze or forced shutdown. KVMSEC implements guest modules that will collect and send information about a possible attack that is underway. The host may then choose the appropriate defensive strategy (as required by R4). Furthermore KVMSEC can create a host-side database containing computed hashes for selected critical path files of VMs (see Section 3.1). A runtime daemon can then recompute the hash values for the monitored files. If a mismatch is found, adequate countermeasures such as those just depicted above, can be taken.

4. KVMSEC IMPLEMENTATION

The KVMSEC project is divided into two major sections (see Figure 5): host and guest. Both have a similar structure, that is:

- A kernel module that manages and shares a communication channel with a userspace module;
- A userspace module that dynamically receives messages from the shared memory, analyzes them and then generates responses.

The communication protocol (see Figure 4) between the various modules (both kernel and user space) is the same. Salient differences between host and guest are:

- Management and allocation of shared memory:
  - In the VM the shared memory is allocated and managed by the kernel module;
  - In the host the shared memory must be already allocated (in the VM) and its management is delegated to Qemu-KVM;
- The number of modules in user space:
  - In the VM we need only one userspace module because all the shared memory management complexity is delegated to the kernel;
  - In the host we need at least two userspace modules because one of them (DM) will deal with the generation and processing of messages exchanged to and from the VM while the other one (Qemu-KVM) manages shared memory and message delivery processes.

4.1 KVMSEC Host side

The host part is composed of three modules:

1. A Kernel Module (KVMSECD) that is activated during the first stages of KVMSEC;
2. A module (DM) residing in user space that elaborates and generate responses to the messages from the VM;
3. A module (Qemu-KVM) residing in user space and managing the shared memory between host and VM; this module delivers the message from the DM module to the VM and vice-versa.

In the following, we describe the above mentioned components, main algorithms, and communication protocols:

KVMSECD: It is a kernel daemon, resides in kernel-space and has access to all VM address space. In addition, this module is aware of the other two host modules (Qemu-KVM and DM) because they will register their pids with KVMSECD. KVMSECD searches into the VM address space, more precisely the one dedicated to Qemu-KVM, a predetermined SEARCH_STRING. Once found, a pointer is returned to the start physical address of the shared memory that will be used for the communication between VM and Host. This information is then sent to DM that will forward it to the Qemu-KVM, precisely to the thread that manages the shared memory inside Qemu.

Communication channel towards DM: Communication between KVMSECD and DM is managed through a combination of character device (called *char_dev*) controlled by

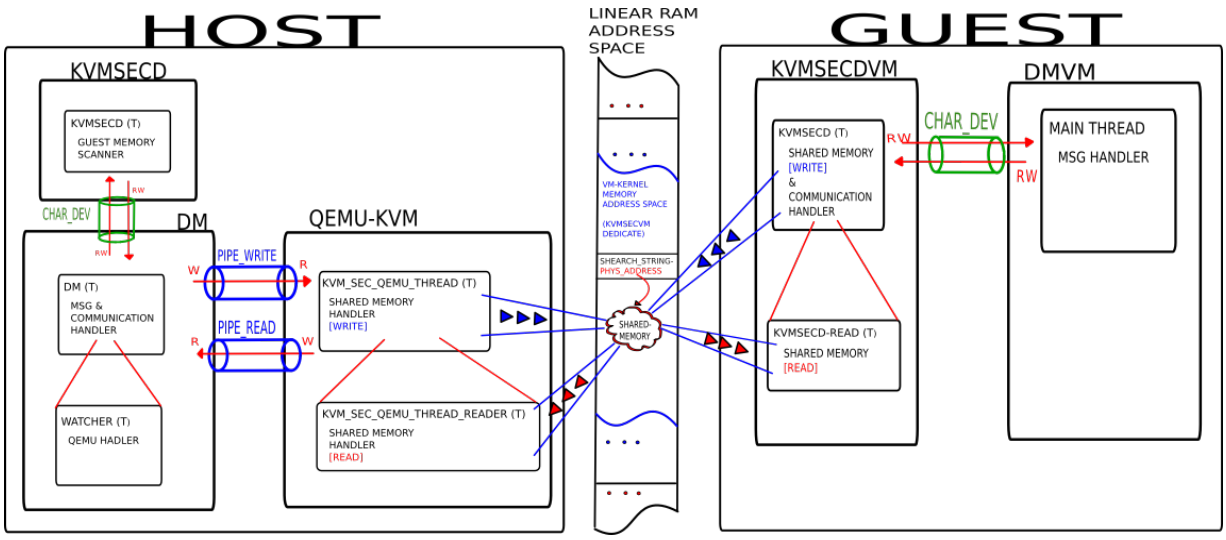

Figure 5: KVMSEC Architecture

the DM through the IOCTL interface and POSIX signals. The capabilities of the character device are extended using the IOCTL interface provided by the system, which allows to define a series of macros that can be used in userspace to interact with the kernel module. In addition, the IOCTL interface, through macros, allows defining the access policy to the device: Read, Write, both or neither of the two. We defined the following macros:

1. Read (IOCTL_GET_MESSAGE) Allows a user process to copy a message from the buffer inside the kernel module;
2. Write (IOCTL_PUT_MESSAGE) Allows a user process to insert a message in the buffer inside the kernel module;
3. NO Read & Write (IOCTL_CONFIRM) Allows a user process to confirm the correct reception/sending of a message;
4. NO Read & Write (IOCTL_REGISTER) Allows a user process to notify the kernel module that the user space daemon is ready to start the registration phase (see Section 4.1).

In every communication phase the core elements (that is, the buffers) are protected by locks inside the kernel module code (for that purpose our code makes use of the following primitives: semaphores *sem_wait*, mutexes: *mux*, atomic variables: *reg* and *qemu_alive*). Access to the buffer by a user process (the DM) is done trough the previously defined macros since the critical section has to be protected by semaphores in the macros. In this way the system is scalable to more than one userspace process, since all access the buffer trough these macros.

Communication between KVMSECD and DM: The communication channel between KVMSEC and DM is an integrated character device (called *char_dev*) from IOCTL as explained above. The adopted signaling mechanism depends on the communication direction: FROM KVMSECD TO DM: The kernel module uses POSIX signals, in particular it sends a SIGUSR1 [15] signal. FROM DM TO KVMSECD: the communication between user space and kernel space is usually considered a critical point for security issues, so we preferred not to register the kthread kpid inside the DM. For the signaling functionality we make use of the previously cited *char_dev* character device. Since we only have to signal an event and we do not need to read or write anything we used the IOCTL macro IOCTL_CONFIRM (as seen in Section 3) that does not access any buffer.

VM memory analysis: When the registration of the two host modules has completed KVMSECD has the Qemu-KVM pid, through which [with the system function find_task_by_pid(int pid)] it finds the structure that describes that process (struct task_struct). From the task_struct of Qemu-KVM, KVMSECD extracts the linked list that contains the pointers to the memory areas currently allocated to that process (struct task_struct → struct mm_struct → struct vm_area_struct). Once obtained such logic memory addresses dedicated to the Qemu-KVM memory areas on the stack (that is, the VM memory), KVMSECD analyzes them to find the kvmsec_string_src and consequently the shared memory physical addresses. After making appropriate checks on the physical address (e.g. that the address is a correct physical address and that it is included in the physical addresses space assigned to Qemu-KVM) KVMSECD can communicate it to the DM that will forward it to the threads in Qemu that manage the shared memory.

First userspace module - DM: DM is the first of two userspace modules and is composed of two threads:

1. DM - This is the main thread that manages:
   (a) The communication between DM and KVMSECD, DM, and Qemu;
   (b) The creation and reception of messages to and

from the shared memory throughout Qemu-KVM;

(c) The registration of the DM pid into KVMSECD;

2. WATCHER - This is a secondary thread that manages:

   (a) The Qemu-KVM startup;

   (b) The registration of the Qemu-KVM pid into KVMSEC;

   (c) The abnormal termination of Qemu-KVM;

Communication channel with Qemu: Since both the DM and the Qemu processes are executed in userspace, we can use any of the Linux System V IPC facilities for Inter-Process Communication. In particular, we make use of a named PIPE (or FIFO) [12]. We create two FIFOs (namely *qemu_pipe_read* and *qemu_pipe_write*). These two are open, respectively, in read mode and in write mode by the DM. They are, respectively, opened in write mode and in read mode by Qemu-KVM to implement a full-duplex channel. The FIFOs are created in the /tmp temporary filesystem area and their life is limited to the DM execution.

Second Userspace module: Qemu-KVM Qemu-KVM [16] is the modified version of Qemu that incorporates the communication mechanism with the kernel module KVM. Changes to Qemu-KVM in the KVMSEC project include the addition of two more threads (namely *KVMSEC_QEMU_THREAD* and *KVMSEC_QEMU_THREAD_READER*), that handle the communication between the DM module and the shared memory. Reads and writes in shared memory are split into two threads. To separate the two functions, writes have been delegated to KVM_SEC_QEMU_THREAD and reads to KVM_ SEC_QEMU_THREAD_READER. To synchronize these two threads, we use a semaphore (mutex).

KVM_SEC_QEMU_THREAD: This is the main KVMSEC thread; its main tasks are:

- Data structure disposal;
- Managing the registration between Qemu and DM;
- Managing Data flows to the shared memory.

KVM_ SEC_QEMU_THREAD_READER: This thread is instantiated from the one above when the pointer to the shared memory is received from DM. The thread periodically checks the shared memory for incoming messages. If a message is found, the thread will pick it off and deliver it to DM, through the *qemu_pipe_read* and *qemu_pipe_write*, cited in the previous paragraph and opened in write and read mode respectively.

Communication channel with VM: The communication channel between the host and the VM is represented by a series of physically contiguous memory locations. The memory is allocated within a kernel module in the VM. The shared memory is composed of three logical locations: the first one is used as a lock to access the other two logical locations; the second one is a buffer for writing; and the last one a buffer for reading.

Communication protocol between Qemu and VM (HOST and VM) : The communication protocol between VM and host relies on the synchronized access to the shared memory area. In order to ensure exclusive access to shared memory, inside critical sections we make use of primitives such as (*sh_mem_read_lock and* sh_mem_write_lock). That is, we use *spinlocks* in the VM to grant exclusive access to the shared memory locks. In the communication between host and guest we can only use spinlocks because these tools are the only synchronization utilities that are available at this level. Note that at host level, synchronization with the VM is simpler because the latter is a standard Linux process; for this reason we can use the usual Linux synchronization facilities. Qemu provides the "cpu_physical_memory_rw" function that allows to write in the VM memory. The host-VM synchronization is built onto this function. In this way multiple access to the two read and write buffers are synchronized and protected; further, these access are also OS-independent. The sequential access to messages is given by message checking in shared memory. Indeed, when one of the two parties attempts to access the shared memory for writing, it first checks if there is a message in the buffer, in that case it will wait the next time slot, otherwise it will write the new message. Reading of a messages implies that the reader empties the buffer, so the next communication party will find it free.

4.2 KVMSEC Guest side

KVMSEC guest consists of two modules:

1. A Linux kernel module (KVMSECDVM) that manages the VM communications with the host. Note that this module is also responsible of the allocation of the shared memory;
2. DMVM: A userspace module that deals with the analysis and creation of responses from/to the VM.

More in details:

KVMSECDVM kernel module: Since this module resides in kernel space it has access rights to kernel memory. The shared memory for communication is allocated by this module. In addition, we allocate a buffer that contains the PHYSICAL address of the shared memory. The KVMSECD-READER kthread performs read accesses from the shared memory. This way it retrieves messages and delivers them to KVMSECD, into the userspace. The communication protocol is identical to the one seen above. During host writings in the shared memory, the execution of KVMSECVM is interrupted. We use spinlocks here to protect the critical section.

DMVM userspace module: This module manages messages to and from the shared memory. The communication channel with KVMSECDVM is a character device (called *char_dev*) as in the host, the communication protocol is the same as the one between DM and KVMSECD. Future implementations of this module will provide plug-ins for integrity checking of messages.

In the following we briefly outline the communication mechanism (see Figure 4) between modules, assuming that process 1 act as a reader and process 2 as a writer:

1. Process 1 must receive a message from process 2. So process 1 will wait the message on a synchronization mechanism (e.g.. semaphore, spinlock);
2. Process 2 composes the message for process 1 and writes it on the buffer of the communication channel (e.g.. FIFO, pipe, shared memory);
3. When the writing process on the communication channel is done process 2 sends a signal to process 1(e.g.. system V signals). Process 2 will wait the acknowledgment of the message from process 1 on a synchronization mechanism;
4. After receiving the acknowledgment signal, process 1 awakes and reads the message from the communication channel buffer;
5. Process 1 sends an acknowledgment signal to process 2.

The communication mechanism between VM and host is similar to the above process. The only difference is that there is no signal channel and, as explained early, there is a periodic check (polling) in the shared memory for the presence of new messages. In the following we show a communication example where the host acts as writer and the guest as reader.

On the host side (see Figure 5):

1. The host generates a message (in the DM user space daemon);
2. The host delivers the message via the shared memory manager (Qemu-KVM);
3. The shared memory manager checks the shared memory for:
   (a) buffer lock not locked;
   (b) buffer empty;

   if the above two conditions are satisfied the host will:
   (a) lock down the lock;
   (b) write the message in the buffer;
   (c) release the lock;

   otherwise the host, at every timed period (5 ms.) will:
   (a) lock down the lock;
   (b) check the presence of message in the buffer;
   (c) release the lock;

In parallel with the host, the guest will:

1. Check for the presence of messages in the shared memory via the kernel module (KVMSECVM in Figure 5); In order to access the shared memory the kernel module must satisfy the following conditions:
   (a) buffer lock not locked;
   (b) buffer empty;
2. If the two conditions above are met the guest will:
   (a) lock down the buffer lock;
   (b) read the message from the buffer;
   (c) release the buffer lock;
3. otherwise the guest at every timed interval (5ms) will:
   (a) lock down the lock;
   (b) check the presence of message in the buffer;
   (c) release the lock;
4. When the kernel module has received the message, it will deliver such message to the guest userspace module (DMVM in Figure 5).

4.3 Simple Use Case

Assume that the entire system is correctly installed on both host and guest machines. Further, assume that a detection engine is present in the host and a data collector is present in the guest. Assume the data collector engine is inside the DM and we have already defined the list of files to be monitored in the guest and a reaction policy in the host. In the following we show a brief example of the entire system in execution.

1. A guest application modifies a file in the critical path;
2. The guest data collector engine notices such alteration since predefined hooks are being involved;
3. The guest data collector engine sends a message via the DMVM thread that manages the communication with the kernel modules KVMSECDVM (see Section 4.1 for the communication between modules);
4. The guest kernel module delivers the message to the host (see 4.1 host-vm communication);
5. The host userspace module (Qemu-KVM) retrieves the message from shared memory;
6. The host userspace module (Qemu-KVM) delivers the message to the daemon (DM);
7. The daemon DM delivers the message to the intrusion detection engine;
8. The intrusion detection engine raises an alarm and notifies the system administrator.

5. DISCUSSION

The goal of KVMSEC is to provide a potentially undetectable and insubvertible system that can detect possible integrity violations of virtual machines. Some characteristics of our work with respect to highlighted requirements are:

- Transparency: KVMSEC guest-host messages do not travel on the network stack (as happens in most integrity architectures, see Section 2.2) so they are virtually undetectable (R1); KVMSEC relies entirely on its own internal communication protocol (see Section 4.1), which makes it independent from the virtualization system, unlike XenRim which relies on services offered by the Xen hypervisor. Furthermore, in KVMSEC each VM has its own reserved shared memory area for guest-host communication which makes each communication channel independently managed and isolated from other channels (R2).
- Full virtualization support: KVMSEC supports *full-virtualization* (see Section 2). As a consequence KVMSEC is potentially undetectable (R1) since no modification at all is required at the guest kernel for running inside the VM.
- Signalling: KVMSEC is potentially less detectable than XenRim since there is no signalling channel at all between host and guest(R1) (see Section 4.1).
- Daemon Process: KVMSEC can share the monitoring task between a kernel module and a user-space module. Distributing the workload should offer performance benefits in normal conditions. The tradeoff is that a guest userspace module might render the whole system more detectable. Anyway the guest userspace module is not required in KVMSEC architecture since its task can be performed by a guest kernel module (see Section 4.1) (R1).
- Resistance to compromising: The core detection system is located on the host side, not on the guest side, thus making the system harder to compromise. In addition, the module that manages host-guest communication is protected inside guest kernel (see Section 4) (R2).
- Deployment: KVMSEC can be deployed on any recent standard Linux kernel whereas e.g.. XenRim requires the Xen virtualization system to be installed and to work on the host machine. As a consequence, the number of supported host platforms is much larger for KVMSEC than for Xen-based solutions. (see Section 2.1) (R3).

6. CONCLUSIONS

In this paper we proposed an extension to the Linux Kernel Virtual Machine. In particular, we extended KVM focusing on security issues, providing a solution (KVMSEC) for real time monitoring of the integrity of the guest machines. To the best of our knowledge this is the first work regarding security inside Linux KVM. The KVMSEC prototype we developed enjoys the following features: it is completely transparent to guest machines (even malicious guest machines); it supports *full-virtualization* which renders the system less detectable on guest side; it can collect data on guest machines; it provides secure two way communications between the hypervisor and guest machine;
it can be deployed on both x86 and x86_64 machines (depends on Linux support). As for futher research directions, we are striving to ease the way IDS modules and Filesystem Integrity Tools can be plugged-in to extend the functionality of the system, fully leveraging the relevant KVMSEC features mentioned above.
Further investigation will also regard performance issues for KVMSEC. Finally it is worth noting that KVMSEC source code is open and freely available on SourceForge: we expect our proposal could increase the interest that the open source community is devoting to security issues in virtualization.

7. REFERENCES

[1] Sgi inc. lkcd - linux kernel crash dump. http://lkcd.sf.net, April 2006.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10, New York, NY, USA, 2006. ACM.

[4] P. Barham, B. Dragovic, K. Fraser, and al. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] G. Collier, D. Plassman, and M. Pegah. Virtualization's next frontier: security. In *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS conference on User services*, pages 34–36, New York, NY, USA, 2007. ACM.

[7] D. Collins. Using vmware and live cd's to configure a secure, flexible, easy to manage computer lab environment. *J. Comput. Small Coll.*, 21(4):273–277, 2006.

[8] R. Di Pietro and L. V. Mancini. *Intrusion Detection Systems*, volume 38 of *Advances in Information Security*. Springer-Verlag, 2008.

[9] J. Dike. User-mode linux. In *ALS '01: Proceedings of the 5th annual conference on Linux Showcase & Conference*, pages 2–2, Berkeley, CA, USA, 2001. USENIX Association.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium,*, February 2003.

[11] Gartner-Group. virtualization to rule server room by 2010. http://www.gartner.com/, 2008.

[12] S. Goldt and al. The linux programmers guide - named pipes. http://www.tldp.org/LDP/lpg/lpg.html, 1995.

[13] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 81–90, New York, NY, USA, 2007. ACM.

[14] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM.

[15] B. Nguyen. *Linux Filesystem Hierarchy (Appendix A)*. The Linux Project, 2004.

[16] Qumranet. Linux kernel virtual machine. http://kvm.qumranet.com.

[17] N. A. Quynh and Y. Takefuji. A real-time integrity monitor for xen virtual machine. In *ICNS '06. International conference on Networking and Services*, pages 90–90. IEEE, 2006.

[18] N. A. Quynh and Y. Takefuji. A novel approach for a file-system integrity monitor tool of xen virtual machine. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 194–202, New York, NY, USA, 2007. ACM.

[19] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283, New York, NY, USA, 2007. ACM.

[20] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: techniques and applications. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36, New York, NY, USA, 2005. ACM.

[21] SWSoft. Virtuozzo linux virtualization. http://www.virtuozzo.com.

[22] T. A. team. Aide: Advanced intrusion detection environment. http://sourceforge.net/projects/aide, November 2005.

[23] T. O. team. Osiris host integrity monitoring. http://www.hostintegrity.com/osiris/, September 2005.

[24] B. Wotring, B. Potter, M. Ranum, and R. Wichmann. *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.

[25] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, New York, NY, USA, 2008. ACM.