Protecting Kernel Code and Data with a Virtualization-Aware Collaborative Operating System

Daniela Alvim Seabra de Oliveira and S. Felix Wu

Computer Science Department, University of California at Davis

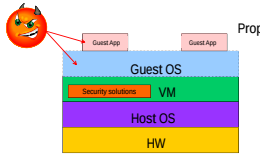
Abstract

The traditional virtual machine usage model advocates placing security mechanisms in a trusted VM layer and letting the untrusted guest OS run unaware of the presence of virtualization. In this work we challenge this traditional model and propose a collaboration approach between a virtualization-aware operating system and a VM layer to prevent tampering against kernel code and data. Our integrity model is a relaxed version of Biba's and the main idea is to have all attempted writes into kernel code and data segments checked for validity at VM level. The OS-VM collaboration bridges the semantic gap between tracing low integrity objects at OS-level (files, processes, modules, allocated areas) and architecture-level (memory and registers). We have implemented this approach in a proof-of-concept prototype and have successfully tested it against 6 rootkits (including a non-control data attack) and 4 real-world benign LKM/drivers. All rootkits were prevented from corrupting kernel space and no false positive was triggered for benign modules. Performance measurements show that the average overhead to the VM for the OS-VM communication is low (7%, CPU benchmarks). The greatest overhead is caused by the memory monitoring module inside the VM: 1.38X alone and 1.46X when combined with the OS-VM communication. For OS microbenchmarks the slowdown for the OS-VM communication was 1.16X on average.

Introduction and Motivation

•VM's extensively used in security-related applications (IDS, secure logging and replay, recovery) due to desirable properties of environment isolation, OS and architecture extensibility; easy manipulation of internal state;

Traditional VM Usage Model
Proposed by Chen and Noble (HotOS'01)

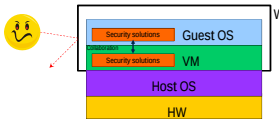


•We challenge this model for security solutions relying on inspecting and manipulating OS abstractions;
- •security solutions operating only at VM level are limited: semantic gap between VM and OS;
- •building secure, transparent and isolated VM is fundamentally infeasible:
  - •malware can present itself as VM (Subvirt – S&P'06, Blue Pill);
  - •malware can detect VM and change its behavior:
    VM's are not designed with transparency as requirement;
    - •Intel x86 architecture is not fully virtualizable (Robin and Irvine – USENIX'00);
    - •virtual and physical environments differ in timing characteristics and hardware configurations.

•In the long run, attackers will operate regardless of the presence of a VM.

Virtualization-aware OS + VM = fine-grained, flexible, stronger system protection for certain security requirements

What are we proposing?



Key Advantage

•We propose an architecture and interface for virtualization-aware OS and VM collaboration in security applications:
- •initial version of architecture and interface for kernel integrity protection:
  - •attacker should not compromise: integrity of collaborating guest OS and information exchanged between OS and VM.

Attacks on Kernel Integrity

•Preventing, detecting and recovering from kernel attacks is difficult: complexity of kernel code, variety of kernel data structures, hard to determine kernel good state, several attacks avenues (vulnerabilities, LKM, /dev/kmem, VM's, hardware, social engineering);

•Proposed solutions for kernel defense:
- •most focus only on detection;
- •prevention approaches:
  - •code authentication: no protection against non-control data attacks;
  - •policies: may not consider all possible data structures.

Main Idea

•Employ a relaxed version of Biba's Integrity model on an architecture where a virtualization-aware OS and a VM can actively collaborate.
- •Biba's * (star) Integrity Axiom: no subject can write objects of higher integrity;
- •two integrity levels: high and low;
- •establishment time:
  - •everything in kernel space immediately after boot is high integrity;
  - •network bytes (including downloaded data) entering system are low integrity:
    network: main vector for attacks and malware;
- •subjects: instructions at architectural level + processes and functions of LKM at OS level.
- •objects: kernel code and data segments at architectural level + files, dynamically allocated kernel memory at OS level.

•All attempted writes into kernel code and data segments are checked for validity at architectural level;

•If low-integrity instruction attempts to write in high integrity areas of kernel space, write is aborted:
- •associated process is terminated and identified, system execution continues, kernel integrity intact.

•Why only Kernel code and Data Segments?
- •Stack segment is expected to be written by low integrity subjects:system calls and kernel exported functions parameters and variables.

Technical Approach

•We need to keep track of low integrity data at OS and architectural level:
- •VM and OS need to collaborate: dual-layer of monitoring bridges semantic gap between OS-VM objects.

Tracking Low Integrity at VM Level

•Shadow memory space: keep track of how network bytes (low integrity data) and their network sources (IP, port) propagate into the system (main memory, register bank, memory for network device).

•Instruction integrity: integrity of memory location where it is stored.

Tracking Low Integrity at OS Level

•File Integrity:
- •when file is written, OS downcalls VM to request information about integrity of bytes;
- •VM checks byte integrity at shadow memory and OS updates file integrity level (inode);
- •implicit low integrity propagation: when file low integrity bytes are read, OS requests VM to mark region as low integrity.

•Process Integrity: integrity is that of executable file and kept in *task_struct;*

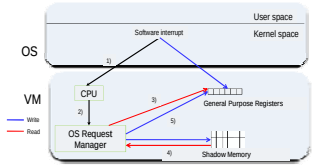
•Module Integrity: OS checks integrity level of file storing object code;
- •if file is low integrity OS downcalls VM to mark memory area holding object code as low integrity.

•Dynamically Allocated Kernel Memory:
- •when low integrity subject allocates kernel memory, OS requests VM to mark area as low integrity:
  - •allows low integrity subjects to write into their own allocated memory;
- •when area is freed, OS downcalls VM to set integrity to high again.

•Interface OS -> VM:
- •software interrupt: unused vector number 15 (reserved by Intel);
- •parameters and return values passed in general purpose registers;

•Interface VM -> OS: exceptions.

OS-VM Communication



Experimental Evaluation

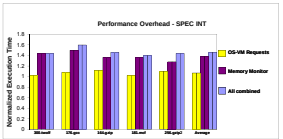
•Proof-of-concept prototype: Linux 2.4.21 (Collaborating Guest OS) + Bochs (VM);

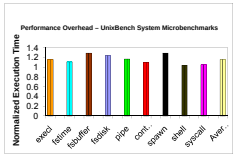
•Metrics: effectiveness of architecture against kernel attacks, behavior of architecture with benign LKM's (false positives) and performance overhead.

Rootkit/Attack	Attack vector	Parts of kernel affected	Non-control data?
adore	LKM	System call table	×
rkit	LKM	System call table	×
knark	LKM	System call table, /proc interface, inet protocols	×
rial	LKM	System call table	×
suckit	/dev/kmem injection	System call table	×
Resource Wastage (Arati et al – S&P'07)	LKM	zone_table	√

Rootkit Attacks

•Benign modules and drivers: ftpfs, FISTGEN, kmw, frandom (sourceforge.net).





Concluding Remarks

•We challenged the traditional VM usage model and proposed a collaboration approach between a virtualization-aware operating system and a VM layer to prevent tampering against kernel code and data;

•Main idea: use a relaxed version of Biba's integrity model and have all attempted writes into kernel code and data segments checked for validity at VM level;

•Proof-of-concept prototype: tested against 6 rootkits (including a non-control data attack) and 4 real-world benign LKM/drivers;

•Performance measurements:
- •average overhead to the VM for the OS-VM communication is low (7%, CPU benchmarks).
- •greatest overhead: VM memory monitoring module: 1.38X alone;
- •OS microbenchmarks 1.16X slowdown for the OS-VM communication.

•Benign modules that modify kernel data structures directly are reported as malicious;

•No protection against: malicious DMA or peripheral devices writes or hardware supported attacks (e.g Cloaker – S&P 2008);

• Research work to appear in ACSAC (Annual Computer Security Applications Conference) 2009.

•Future work (Opportunities for technology transfer):
- •demonstrate how this architecture can benefit other types of security applications;
- •investigate how multi-core technology could decrease the memory monitor overhead;

•Contact: {oliveira, wu}@cs.ucdavis.edu

•This work has been supported by grants FA9550-07-1-0532 (AFOSR MURI) and 0335299, 0520269, 0627749 (NSF).

www.PosterPresentations.com