Ianus: Secure and Holistic Coexistence with Kernel Extensions - A Immune System-inspired Approach

Daniela Oliveira  Jesus Navarro  Nicholas Wetzel  Max Bucci
Bowdoin College, Brunswick ME  NVIDIA Santa Clara CA USA
{doliveir, nwetzel, mbucci}@bowdoin.edu  jnavarro@nvidia.com

ABSTRACT
Kernel extensions, especially device drivers, make up a large fraction of modern OS kernels (approximately 70% in Linux). Most extensions are benign and represent a convenient approach for extending the kernel functionality and allowing a system to communicate with an increasing number of I/O devices. A small fraction of them are malicious and, as they run in kernel space, pose a threat to kernel integrity. From a security viewpoint this situation is paradoxical: modern OSes depend and must co-live with untrustworthy but needed extensions. Our immune system faces the same challenge: our body is made of a large number of bacteria, which are mostly benign and also carry out critical functions for our physiology. However, a small fraction of them pose a threat to our body as they can cause pathologies. The immune system maintains an homeostatic relationship with its microbiota by minimizing contact between bacteria and cell surfaces and confining bacteria to certain sites. Challenging the current trend that advocates leveraging only a hypervisor to defend the kernel (for considering it too vulnerable to defend itself), this paper advocates that modern OSes, like our immune system, should play an active role in maintaining healthy and safe interactions with their extensions. This work presents Ianus, a proof-of-concept prototype for this paradigm using Linux and the Bochs x86 emulator, which successfully minimized kernel extensions interactions with original kernel. Its security was evaluated with real rootkits and benign modules. Ianus’ performance was analyzed with system and CPU benchmarks and it caused an small overhead to the system (approximately 12%).

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Virtual Machine security and protection

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2014 ACM 978-1-4503-2469-4/14/03 ...$15.00.

General Terms
Security

Keywords
Imune system, semantic gap, introspection, collaboration

1. INTRODUCTION

Kernel extensions, especially device drivers, make up a large fraction of modern kernel code bases (approximately 70% in Linux and a larger percentage in Windows) [22]. Most of them are benign and represent a convenient approach for extending the kernel functionality and allowing a system to communicate with an increasing number of complex and diverse I/O devices without the need to reboot the system or recompile the kernel. A small fraction of kernel extensions are malicious (rootkits) and represent a dangerous avenue for system exploitation as they run in kernel mode with complete control of system resources, kernel code and data structures. This situation is paradoxical from a security point of view: modern OSes depend and must co-live with untrustworthy but needed extensions.

Our immune system faces the exact same challenge every day. The human body has ten times more microbes than human cells. Most of these microbes are benign and carry out critical functions for our physiology, such as aiding digestion, preventing allergic reactions, and even controlling obesity [18]. In spite of that, a fraction of these microbes pose a threat to our bodies as they can cause pathologies. Our immune system has evolved so that it can maintain an homeostatic relationship with its microbiota, and this involves controlling microbial interactions with host tissues, lessening the potential for pathological outcomes [18].

This paper advocates that OSes should also evolve so that they play an active role in maintaining healthy and safe interactions with their extensions. Similar to our immune system, the OS should control the interactions between extensions and the original kernel code and data lessening the potential for security breaches. The immune system controls its microbiota using two main approaches: (i) minimization of direct contact between bacteria and cell surfaces and (ii) confinement of penetrant bacteria to certain sites (e.g., the intestine) limiting their exposure to other sites. These approaches are enforced by a set of proteins that bind or engulf bacteria violating the immune system security policies.

In this work a similar paradigm for OS defense is proposed, where the OS plays an active role in controlling the execution of its extensions. The following security policies are adopted: (i) kernel extensions should only interact with the kernel through its exported functions, and (ii) kernel extensions should never directly write into kernel code and data segments, except into their own
memory (module address space and its dynamically allocated areas). Enforcing these policies requires hypervisor support so that modules’ execution are monitored and stopped in case of a violation. Controlling the execution of kernel modules lies at the heart of the proposed approach, which relies on: (i) knowledge of boundaries of kernel modules in main memory, (ii) monitoring of write operations in kernel space, (iii) knowledge of the origin of the current instruction being executed at the CPU, and (iv) knowledge of the addresses of functions invoked by modules. The OS kernel performs task (i) and has knowledge about (iv), but only the architectural layer or a hypervisor can support tasks (ii) and (iii).

This immune-system inspired kernel defense approach challenges the traditional paradigm for OS protection introduced by Chen and Noble in 2001 [7] that advocates placing security and monitoring mechanisms in a hypervisor layer and leaving the OS with no active role in its own monitoring and protection, for it is considered untrustworthy. The threat model is that the hypervisor is trustworthy and the guest OS can be easily compromised by malware. Current virtualization-based security solutions [15, 14, 32, 30, 26] do not count on any collaboration with the guest OS. This traditional model suffers from two main weaknesses.

The first weakness is the semantic gap: there is significant difference between the high level abstractions observed by the guest OS and the low level abstractions at the hypervisor layer. The semantic gap hinders the development and widespread deployment of virtualization-based security solutions because these approaches need to inspect and manipulate abstractions at the OS and architecture level to function correctly. To address the semantic gap challenge, traditional hypervisor-based security solutions use a technique called introspection to extract meaningful information from the system they monitor [15]. With introspection, the physical memory of the guest OS is mapped at the hypervisor address space for inspection. High level information is obtained by using detailed knowledge of the OS layout, algorithms and data structures [11].

The second weakness of traditional hypervisor-based OS defense mechanisms is the introspection mechanism itself, which is a manual, error prone, and time consuming task that, despite being perceived as secure until recently, does rely on the integrity of the guest OS to function correctly. Traditional introspection solutions assume that even if the guest OS is compromised, their mechanisms and tools, residing at a lower-level (hypervisor) will continue to report accurate results. However, Baram et al [4] argued that this security assumption does not hold because an adversary, after gaining control of an OS (e.g., through kernel-level rootkits), can tamper with kernel data structures so as a bogus view of the system is provided for introspection tools.

A proof-of-concept prototype for this new paradigm, called Ianus, was designed and implemented using the Bochs Intel x86 emulator [43] as the hypervisor and Linux Ubuntu 10.04 (kernel version 2.6.32) as guest OS. Ianus’ experimental evaluation showed it successfully confined modules into their own address spaces and contained their interactions with other parts of kernel code and data. Ianus’ security was assessed with a set of real kernel rootkits which were stopped before any malicious actions were performed and with benign modules that could run normally. The overhead to the system was analyzed with a set of system and CPU benchmarks and was found to be low, approximately 12%.

The rest of the paper is organized as follows. Section 2 describes Ianus’ design and implementation. Section 3 details the experimental evaluation performed concerning Ianus’ security and performance. Section 4 discusses the state-of-the-art in virtualization introspection, kernel defense, and module isolation. Finally, section 5 concludes the paper.

2. DESIGN AND IMPLEMENTATION

The immune system-inspired kernel defense approach involves a virtualization-aware guest OS that directly downcalls a hypervisor to pass information about loading, unloading and memory boundaries of modules. Upon being processed by the CPU these downcalls are forwarded to handlers at the hypervisor, which are also responsible for maintaining the information passed. Figure 1 shows Ianus’ high level view, which has the following key features: OS downcalls, downcall processing handlers, modules’ information kept in the hypervisor, and a checker for memory writes and function calls.

Downcalls are direct calls from the OS to the hypervisor (Step 1 in Figure 1) and can have a variable number and type of parameters. Ianus has downcalls for module loading, unloading, and dynamic allocation and deallocation of kernel memory. Every time a module is loaded, the OS downcalls the hypervisor to pass the module’s name and the address and size of its object code. The module’s name uniquely identifies it in the system. When a module allocates memory, the OS downcalls the hypervisor passing information about the address and size of the area. Memory dynamically allocated by modules, despite being part of the kernel, do not receive the same level of protection given to original kernel data areas. The security policy adopted is to not allow kernel code and data being overwritten (bypassing kernel exported functions) by module’s instructions, which are considered low integrity. However, modules cannot be prevented from writing into their own allocated memory areas, which requires the hypervisor to keep track of them. When a module frees memory, the OS downcalls the hypervisor to provide it with the memory region address. Memory deallocated by a module is considered again a high-integrity area of kernel space.

Upon receiving a downcall the CPU delegates its processing to specific handlers (Steps 2 and 3 in Figure 1), which create objects representing modules and their attributes in the hypervisor. Modules’ memory boundaries (a range of linear addresses) are kept in an internal hash table per module in the hypervisor. When a module is unloaded the handler destroys the corresponding module’s object and removes the module’s corresponding linear addresses from the module’s hash table. Whenever kernel memory is allocated the handler checks if the instruction performing the allocation belongs to any of the modules and if it does, the handler inserts this area into the module’s hash table. Finally, when kernel memory is deallocated, the handler checks if the area belongs to any of the
modules active in the system and if it does, this memory region is removed from the module’s hash table of linear addresses.

2.1 Assumptions and Threat Model

This paradigm assumes an active OS which, like the immune system, is in charge of its own protection against its community of modules with the support of a trustworthy hypervisor for tasks that need to access architectural abstractions. It is assumed that most kernel extensions are benign, but a small fraction of them will attempt to compromise the kernel and execute malicious actions.

It is also assumed an establishment time immediately after boot and all code and data present in the system before it are considered trustworthy. All modules installed in the system are monitored, but they do not suffer any restriction on their execution, as long as they do not attempt to bypass kernel exported functions and write into the kernel code and data segments.

2.2 Implementation

A proof-of-concept prototype, Ianus, was implemented to evaluate the proposed immune-system inspired kernel defense approach. Ianus used the Bochs x86 32-bit emulator as the hypervisor and Linux Ubuntu 10.04 kernel version 2.6.32 as the guest OS. Bochs was used due to its flexibility for performing architectural changes; the goal was to show that the proposed approach is promising in providing a secure symbiotic relationship between an OS kernel and its extensions. A high performance hypervisor can also be modified to support such paradigm. The modifications in the guest OS were minimal: a total of seven downcalls were inserted and each one contributed to one single assembly instruction added to the kernel. Bochs was extended with downcall processing handlers, a new vector for the interrupt instruction (downcall), data structures for keeping module’s attributes and range of memory areas, and a checker invoked in all functions performing writes in main memory.

2.2.1 OS Downcalls and Hypervisor Handlers

The downcalls are implemented as unused software interrupts in the Intel x86 architecture (unused vector 15). The INT n instruction generates a call to the interrupt or exception handler specified by the destination operand. This operand (the interrupt vector) is an 8-bit number from 0 to 255, which uniquely identifies the interrupt. Vector 15 is reserved by Intel and is not in use. The INT n instruction was modified to handle vector 15 and this new software interrupt is handled similarly to how system calls are processed, with parameters passed in general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, and EBP).

The downcalls were also placed into the kernel functions kmalloc() and vmalloc() to handle memory dynamically allocated by modules. The addresses of the callers of these allocation functions were obtained using the __builtin_return_address gcc hack to the Linux kernel [1], and allowed the hypervisor handlers to discover whether the caller function belonged to any of the active modules. This strategy allows the distinction between memory allocations made by a module and by the original kernel. If the address of the caller belongs to any of the modules, the handler adds this newly allocated memory region to the module’s hash table. Downcalls were also inserted in the corresponding deallocation functions kfree() and vfree(). The corresponding downcall handlers check whether the caller’s address belongs to any of the active modules tracked and if it does, remove the freed memory range from the module’s hash table.

The downcall handlers at the hypervisor must translate virtual addresses from the OS into linear addresses. Each virtual address is represented by a segment and an offset inside this segment. The virtual addresses are included in the machine language instructions to specify the address of an operand or instruction. For example, in the assembly instruction MOVE EDI, [EBX], the content of memory location given by register EBX is stored into register EDI. In this case, register EBX contains the offset of a virtual address in a particular segment. However, the security mechanisms at the hypervisor deal with linear addresses. In the Intel x86 architecture (used in this work) a linear address is a 32-bit number used to address a memory range up to 4 GB (addresses 0 to 2^{32} – 1).

Linux employs a limited form of segmentation by using three registers (cs, ds, and ss) to address the code (CS), the data (DS) and the stack (SS) segments. Processes running at user-level mode use these registers to address respectively the user code, data and stack segments. Code executing at kernel-level use these registers to address the kernel data, code and stack.

Each handler, upon receiving a virtual address from the OS in one of the general purpose registers must translate it into a linear address. The virtual address (segment and offset) is forwarded to the segmentation unit in the hypervisor and translated into a 32-bit linear address that can be used to index the modules’ hash table. Downcalls passing memory addresses can refer to data structures stored in the kernel data segment or code in the kernel code segment. For instance, the name of a module or the address of a dynamically allocated memory region are located in the kernel data segment. A module’s core and init parts reside in the kernel code segment.

2.2.2 OS and Downcall Integrity

A key requirement of the proposed approach is to guarantee the integrity of the OS downcalls. A kernel module does not have privileges to issue downcalls and should not tamper with downcalls issued by the original kernel. This policy prevents a malicious module from issuing a bogus downcall, or tamper with information passed by the original kernel to the hypervisor. These goals are addressed through the verification of all writes into kernel code and data segments at the origin of a downcall instruction.

The first security policy adopted is that kernel modules are not allowed to perform write operations into the kernel code and data segments. The hypervisor contains a module for checking the validity of all write operations performed in the kernel code and data segments using Algorithm 1. This check is performed immediately before an instruction attempts to write a value into a memory location (mAddr). The hypervisor functions that perform writes into memory (a total of six functions) were instrumentation to invoke the checker before any write is performed in main memory.
Whenever a write operation is about to be performed, it is checked whether the write is being attempted at kernel mode. This is done by checking the CPL (current privilege level) value, which is represented by a 2-bit field in the ea register. Then it is checked whether the linear address of the instruction storing data (iAddr) into memory belongs to any of the module’s memory region monitored at the hypervisor. Next, it is checked whether the memory address being written (mAddr) belongs to the module itself, which is not considered a security violation. Following, the segment being written is checked. If the write is attempted at the data or code segments, an exception is raised because it is considered a security violation (Step 4 in Figure 1). If the segment is SS it is checked whether the target address is higher than the current value of register EBP. If it is higher, this is an indication of a stack corruption attempt and an exception is raised. The kernel has the discretion to treat this policy violation the way it finds most appropriate. One possible action is to unload the offending module from the system.

The integrity of downcall instructions is checked with hypervisor support. Upon execution of the INT $15 instruction it is checked whether the instruction bytes come from a module. This is done by hashing the current instruction linear address to the hash tables that maintain the memory regions for modules. If the instruction belongs to any of the modules, an exception is raised.

### 2.2.3 Monitoring Non-Exported Function Calls

Ianus monitors extensions interactions with kernel functions. Extensions are supposed to only invoke kernel exported functions. Ianus intercepts all CALL instruction invoked by a module and checks whether its target address belongs to an exported function. If the target address of the CALL instruction corresponds to a non-exported function (from the kernel or other extensions) or even to an address that does not correspond to a function (an indication of a return-oriented attack [33]), Ianus raises an exception.

The addresses of all kernel functions (exported and non-exported) are obtained from the System.map file created during kernel compilation and made available to the hypervisor.
Linux versions attempting to make the system call table inaccessible to kernel extensions. The rootkit has keylogging capabilities and was based on the system call hijacking approach described in [38]. The rootkit hijacks the system call table by first locating its address in the kernel through brute force and writes into the system call table by first setting the CR0 register’s first bit to 0, which changes the CPU from protected to real mode. After tampering with the system call table, the rootkit puts the CPU back to protected mode. These actions were done with the kernel functions read_cr0 and write_cr0.

When this malicious module is loaded, the hypervisor has complete information about its boundaries in memory. When the initialization function is invoked, one of its instructions attempts to perform a write operation in an area which is part of the kernel data segment. The goal is to overwrite this area with a malicious address into one of the slots of the system call table. The write operation is checked at the hypervisor level and it is detected that (i) it is performed in kernel mode, (ii) the target memory location is in the kernel data segment, (iii) the instruction bytes come from the text segment of the module’s init part, and (iv) the memory area being written is not part of any module’s dynamically allocated memory region. The write operation is aborted (thus preventing any compromise) with the hypervisor raising an exception handled by the OS. All other rootkits that operate by tampering with the system call table were stopped similarly.

**Issuing a Bogus Downcall:** Here the goal was to evaluate whether or not kernel-level malware could issue bogus downcalls to the hypervisor. The authors implemented a rootkit that attempted to perform a downcall passing fabricated parameters to the hypervisor. The downcall was issued in the module’s initialization function. As in the previous examples, immediately before the initialization function is invoked the hypervisor is keeping track of all memory areas in use by this module. The module’s initialization function is invoked and issues a downcall causing the CPU to execute instruction INT $15. Upon executing the interrupt instruction the origin of its bytes is verified at the hypervisor by hashing the instruction linear address to the hash tables that maintain the modules’ memory regions. The hash is a hit, which shows that the downcall is being issued by a module, and an exception is raised.

The only rootkit Ianus was not able to contain was the kbd_notifier keylogger [2], which operates without the need to tamper with kernel code and data. It is a stealthy rootkit that works by registering a malicious function with the kernel keyboard notifier chain, which is invoked whenever a keyboard event occurs and allows the malware to record the keys pressed by a user at the kernel level.

**Benign Kernel Extensions:** Another important point in the evaluation was to analyze Ianus’ behavior when executing benign kernel modules. The assumption is that benign modules will only access kernel exported functions to perform their tasks. Table 2 illustrates the evaluation done with benign drivers installed during boot and benign extensions from SourceForge [44]. From the set of extensions analyzed, three benign drivers invoke non-exported functions from other modules and the kernel. These issues caused Ianus to raise an exception to the OS. However, the system can be configured to ignore security issues raised by modules known to be benign.

Attacks that do not need to write into kernel space to succeed [10] or that compromise a hypervisor [23, 31] or that write into kernel memory through trusted code paths in the kernel are beyond the scope of Ianus. Further, the target of JMP instructions, which can be leveraged by a rootkit to bypass Ianus’ checks, are not currently checked.

**3.2 Performance Analysis**

This section analyzes Ianus’ performance impact in the whole system using system microbenchmarks from Unixbench [39] and a subset of the SPEC CPU INT2006 benchmark suite [45]. The execution times were normalized to the execution time of the system without any modifications to the OS and the Bochs x86 emulator. Using the unmodified Bochs as a basis for normalization allowed the evaluation to be focused on the actual overhead of the security mechanisms and not on the Bochs overhead as an Intel x86 emulator.

Figure 2(a) shows the performance overhead of the OS downcalls during normal OS operation for Unixbench. These benchmarks exercised exec calls, file reads and writes (fdbuf, fsdisk and fstat), pipe throughput (pipe) and pipe-based context switch (context1), process creation (apsem) and system call overhead (syscall). For these experiments Unixbench ran with the modified and the unmodified version of the guest OS. The goal here was to isolate the performance overhead of downcall issuing at the OS for intense system-level activity. Figure 2(a) shows that the overhead of downcall issuing at the OS is negligible (average 2%) for most of the system benchmarks.

Figure 2(b) shows the performance analysis for five benchmarks from SPEC CPU INT2006. The overhead was measured for two different situations. In the first (OS Downcalls), the system was running the modified version of the OS containing all downcall issuing. Here the goal was to evaluate the overhead to the OS for a CPU intensive benchmark. The second setting (Downcall handling) had the same configuration as the first, but now the downcalls were being processed by the hypervisor’s handlers. Figure 2(b) corroborates Figure 2(a) results in the sense that the downcall overhead at the OS is very low. Downcall processing caused an increase of 12% on average to the execution time of the SPEC CPU INT benchmarks at the hypervisor. The overhead of 12% is low when we consider that certain types of applications that require a high level of security, (e.g., a power grid or a server at a security agency), can trade performance for security. As future work, we plan to analyze Ianus’ memory overhead.

<table>
<thead>
<tr>
<th>Name</th>
<th>Attack approach</th>
<th>Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>kBeast</td>
<td>system call hooking</td>
<td>network activity, process/file hiding, keylogging, anti-delete/kill/remove</td>
</tr>
<tr>
<td>lDDUta</td>
<td>system call hooking</td>
<td>covert channel, module, process and file hiding, keylogging and privilege escalation</td>
</tr>
<tr>
<td>LIIES</td>
<td>system call hooking</td>
<td>keylogging</td>
</tr>
<tr>
<td>rkit</td>
<td>system call hooking</td>
<td>privilege escalation</td>
</tr>
<tr>
<td>kbd_notifier</td>
<td>registration of malicious function with notifier chain</td>
<td>keylogging</td>
</tr>
<tr>
<td>Bogus Downcall</td>
<td>direct invocation of INT $15</td>
<td>issuing of a bogus downcall to the hypervisor</td>
</tr>
<tr>
<td>General Keylogger</td>
<td>system call hooking</td>
<td>keylogging</td>
</tr>
</tbody>
</table>

Table 1: Rootkits used in the security evaluation.
Figure 2: Performance overhead.

<table>
<thead>
<tr>
<th>Module</th>
<th>Non-exported functions invoked</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c_piix4</td>
<td>none</td>
</tr>
<tr>
<td>serio_raw</td>
<td>none</td>
</tr>
<tr>
<td>floppy</td>
<td>none</td>
</tr>
<tr>
<td>parport_pc</td>
<td>add_dev(parport)</td>
</tr>
<tr>
<td></td>
<td>parport_device_proc_register(parport)</td>
</tr>
<tr>
<td>parport</td>
<td>parport_pc_data_forward(parport_pc)</td>
</tr>
<tr>
<td></td>
<td>_ticket_spin_unlock(kernel)</td>
</tr>
<tr>
<td></td>
<td>parport_pc_restore_state(parport_pc)</td>
</tr>
<tr>
<td></td>
<td>parport_pc_frob_control(parport_pc)</td>
</tr>
<tr>
<td></td>
<td>parport_pc_read_status(parport_pc)</td>
</tr>
<tr>
<td></td>
<td>lp_attach(lp)</td>
</tr>
<tr>
<td></td>
<td>pp_attach(ppdev)</td>
</tr>
<tr>
<td>psmouse</td>
<td>none</td>
</tr>
<tr>
<td>ppdev</td>
<td>none</td>
</tr>
<tr>
<td>lp</td>
<td>none</td>
</tr>
<tr>
<td>8390</td>
<td>_ticket_spin_unlock(kernel)</td>
</tr>
<tr>
<td></td>
<td>ne2k_pci_get_8390_hdr(ne2k_pci)</td>
</tr>
<tr>
<td>ne2k_pci</td>
<td>none</td>
</tr>
<tr>
<td>Benign modules from sourceforge</td>
<td></td>
</tr>
<tr>
<td>frandom</td>
<td>none</td>
</tr>
<tr>
<td>tier</td>
<td>none</td>
</tr>
<tr>
<td>tty0tty</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2: Modules’ access to functions in kernel space (benign modules).

4. RELATED WORK

A great amount of research has been done regarding hypervisor-based intrusion detection systems and security solutions leveraging traditional introspection. This section discusses the state-of-the-art in hypervisor-based introspection and kernel integrity defenses.

**Hypervisor or Virtual Machine (VM) Introspection:** VM-based intrusion detection systems leverage introspection in two ways: passive [15, 19] and active [26, 28, 3]. Passive introspection accesses a guest OS memory to reconstruct its state and abstractions. The OS state is recreated from low level data such as memory page frames.

Active introspection addresses better the semantic gap problem by allowing a more up-to-date view of a guest OS state. Xenprobes [28] and Latres [26] place hooks inside the guest OS to intercept some key events, and invoke a security tool residing at VM level to treat the event. HIMA [3] is a VM-based agent to measure the integrity of a guest OS by intercepting system calls, interrupts and exceptions. All of these approaches (passive and active) consider the guest OS untrustworthy and do not actively interact or leverage it in the introspection process. This limits the amount and variety of system-level information that can be collected. L4 microkernels ([24]) functioning as a hypervisor also requires a L4-aware OS, which resembles the approach proposed here that leverages a virtualization-aware OS. The OS modifications turn system calls, memory and hardware accesses into calls to the hypervisor. Differently from L4 microkernel, the purpose of the OS downcalls in Ianus is exclusively for aiding security.

Recently, researchers have been working on better ways to perform traditional introspection. Chiueh et al [9] inject stealthy kernel agents to a guest OS to enable virtual appliance architectures to perform guest-OS specific operations. Virtuoso [11] creates introspection tools for security applications with reduced human effort. SIM [34] enables security monitoring applications to be placed back in the untrusted guest OS for efficiency. It still suffers from the same semantic gap challenges as traditional introspection approaches because it was not designed to rely on data from the guest OS. Fu and Lin [12] apply system-wide instruction monitoring to
automatically identify introspection data and redirect it to the in-guest kernel memory. A limitation is that certain types of data cannot be redirected, limiting the amount of guest OS information that can be obtained. Other line of work [35, 16], based on process migration, proposes to relocate a suspect process from inside the guest OS to run side by side with an out-of-VM security tool. The challenge is that some processes are not suitable for migration.

**Kernel Integrity Defense:** Many authors have previously addressed kernel protection. Those focusing on prevention use some form of code attestation or policy to decide whether or not a piece of code can be executed in kernel mode. SecVisor [32] employs a hypervisor to ensure that only user-approved code executes in kernel mode: users supply a policy, which is checked against all code loaded into the kernel. NICKLE [30] uses a memory shadowing scheme to prevent unauthorized code from executing in kernel mode. A trusted VM maintains a shadow copy of the main memory for the guest OS and performs kernel code authentication so that only trusted code is copied to the shadow memory. During execution, instructions are fetched only from the shadow memory. Code attestation techniques [14] verify a piece of code before it gets loaded into the system.

Some approaches can offer some protection against non-control data attacks [8] that tamper with kernel data structures by directly injecting values into kernel memory. Livewire [15] is a VM architecture with policies for protecting certain parts of the kernel code section and the system call table. KernelGuard [29] prevents some dynamic data rootkit attacks by monitoring writes to selected data structures. Oliveira and Wu [25] used a performance expensive dynamic information flow tracking system (DIFT) and a set of shadow memories to prevent untrusted bytes to reach kernel space.

There are also many works addressing detection. Copilot [27] uses a PCI add-in card to access memory instead of relying on the kernel. Lycosid [21] and VMWatcher [20] perform detection based on a cross-view approach: hiding behavior is captured by comparing two snapshots of the same state at the same time but from two different points of view (one from the malware and the other not). OSck [17] protects the kernel by detecting violation in its invariants and monitoring its control flow.

The difference between this work and previous research in VM introspection (passive and active) and OS protection is that here the OS, like our immune system, has an active role in its protection against compromise from kernel extensions. The hypervisor acts only as a supporting layer leveraging the key information about modules passed by the OS to monitor the interactions of modules and the original kernel. Having the OS in charge of monitoring itself streamlines kernel defense when compared to related work based on manual and error-prone introspection and separate hardware for defense.

**Module isolation**

A great body of work in the literature focus on isolating or con-finishing the effects and execution of device drivers and modules. Nooks [37] introduced the concept of shadow drivers and isolate them in a separate address space so that they can be recovered after a fault. HUKO [42] and Gateway [36] built on this idea by leveraging hypervisor support to protect the approach that confine modules in a separate address space from a compromised OS. Ianus’s goals are similar to Nooks, HUKO, Gateway in the sense of protecting the kernel from malicious or misbehaving extensions. However, Ianus does not attempt to physically isolate the kernel from its extensions, but provide a way for them to co-exist. This provided much more flexibility to the system. For example, Section 3 showed that many drivers do invoke kernel (and other module’s) non-exported functions and their execution would be disrupted in HUKO or Gateway. Ianus can be fine tuned to allow more privileges to certain modules known to be benign.

Some lines of work advocate running drivers partially or entirely in user space. Ganapathy et al [13] introduced the idea of micro-drivers in which drivers execute partly in user space and partly in the kernel. Nexus [41] and SUD [6] confine buggy or malicious device drivers by running them in user-level processes. Some works attempt to achieve driver isolation in software, such as SFI [40], where the object code of untrusted modules are rewritten to prevent their code from jumping to an address outside of their address space. Ianus allows extensions to be executed without any modifications.

5. CONCLUSIONS

This paper introduced a paradigm for OSes to safely co-exist with their untrustworthy but needed extensions that is based on the immune system relationship with its microbiota. Both kernel and immune system face the same paradoxical challenge: most kernel extensions and bacteria are benign and crucial for carrying out important functions in the system, but a small fraction of them pose security or healthy threats; malicious extensions compromise the entire system as they run with kernel privileges, while malicious microbes can cause serious pathologies. This paper advocates employing containment similar approaches similar to those developed by the immune system to allow an OS kernel to safely co-exist with its extensions: prevent extensions from directly modifying kernel code and data segments, except when they leverage exported kernel functions.

A proof-of-concept prototype for this approach, named Ianus, was implemented with Linux and the Bochs x86 emulator as the supporting hypervisor. Ianus’s was studied with several real rootkits and benign extensions. In the experiments all malicious rootkits were stopped and Ianus caused no false positives for benign modules. Ianus’ performance was analyzed with system and CPU benchmarks and the system overhead was low (12% on average).

**Acknowledgments**

We would like to thank the SAC anonymous reviewers for valuable feedback. This research is funded by the National Science Foundation under grant CNS-1149730.

6. REFERENCES


[38] Syscall Hijacking (http://memset.wordpress.com/2011/03/18/syscall-hijacking-dynamically-obtain-syscall-table-address-kernel-2-6-x-2-)).


