Ensuring Operating System Kernel Integrity with OSck

Owen S. Hofmann Alan M. Dunn Sangman Kim Indrajit Roy* Emmett Witchel

The University of Texas at Austin *HP Labs

{osh,adunn,sangmank,witchel}@cs.utexas.edu indrajitr@hp.com

Abstract

Kernel rootkits that modify operating system state to avoid detection are a dangerous threat to system security. This paper presents OSck, a system that discovers kernel rootkits by detecting malicious modifications to operating system data. OSck integrates and extends existing techniques for detecting rootkits, and verifies safety properties for large portions of the kernel heap with minimal overhead. We deduce type information for verification by analyzing unmodified kernel source code and in-memory kernel data structures.

High-performance integrity checks that execute concurrently with a running operating system create data races, and we demonstrate a deterministic solution for ensuring kernel memory is in a consistent state. We introduce two new classes of kernel rootkits that are undetectable by current systems, motivating the need for the OSck API that allows kernel developers to conveniently specify arbitrary integrity properties.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Invasive software

General Terms Security, Verification

Keywords Rootkit detection

1. Introduction

Rootkits are software packages left on an exploited system to facilitate further malicious access. They compromise operating system security in ways that are difficult for modern tools to detect. Increasingly dangerous are *kernel rootkits* that use access to kernel memory to subvert the integrity of kernel code or data structures, giving control of critical resources (such as processing, the file system, or the network) to a malicious entity [29].

Kernel rootkits are difficult to detect because operating systems have complicated data structures with no explicit integrity constraints, providing a large and growing attack surface to rootkit authors. In addition, user- or kernel-level security tools should not trust a compromised operating system kernel to correctly execute security checks. To efficiently and reliably detect rootkits, we propose OSck[1], a hypervisor-based system to specify and enforce the integrity of operating system data structures. Kernel rootkits change the state of operating system data structures in order to gain unauthorized access to computer resources and to prevent detection. OSck detects when the state of kernel data structures violates the integrity properties specified to the hypervisor. If a rootkit compromises kernel integrity, the hypervisor can take appropriate action, such as terminating the operating system or alerting an administrator.

We extend previous work in hypervisor-based monitoring in four important directions:

1. OSck verifies type-safety properties for the kernel heap through a linear scan of memory, rather than traversing a data structure graph. This approach is based on extracting assumptions about kernel memory layout from memory management data structures. It is more efficient than graph traversal in both time and space, and facilitates incremental verification.
2. OSck correctly handles concurrency between the running kernel and verification process. Previous systems assumed that any inconsistencies caused by concurrency would be transient. We show this assumption incorrect through user code that causes these errors frequently enough so as to appear persistent.
3. OSck exports an expressive API for specifying ad-hoc integrity checks. With OSck, integrity checks are written as if the code executes in a kernel thread, with full access to all kernel data structures, even though the code executes in the context of an unprivileged hypervisor address space.
4. We design and implement two new classes of rootkits and implement new versions of known attacks, demonstrating that defeating new rootkit exploits with OSck is simple, often requiring fewer than 100 lines of code. One of our new attacks would evade detection by all previous monitoring systems because it modifies data that is not reachable by traversing kernel data structures from global roots.

OSck supports a rich API for writing kernel integrity checks. We give three examples of integrity properties that vary in complexity and require different levels of support from OSck. First, the dbgreg rootkit sets up the x86 debugging registers to transfer control to it when the kernel executes the code to demultiplex a system call. OSck prevents this rootkit's operation simply by verifying that x86 hardware debug registers do not reference kernel text. Second, most in-the-wild kernel rootkits modify kernel state by directing function pointers to custom functions [23]. OSck detects such rootkits by verifying type-safety for kernel data structures—the property that each pointer points to an object of the correct type. This is a complex property that must be checked concurrently with kernel execution, and OSck extracts rules about data structure allocations from the kernel source. Third, integrity constraints may require coordination between the kernel and hypervisor to prevent false positives due to race conditions. For instance, the Linux kernel maintains both a tree and list representation of the run queue. Rootkit tasks can avoid being listed by system utilities like `ps` by removing themselves from the list representation, while remaining in the tree representation allows them to be scheduled. However, because

[1] Like `fsck`, but for operating system state.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03…$10.00

the kernel updates these data structures on every scheduling event, workloads that involve significant scheduling activity may require synchronization to coordinate the OS modification with the hypervisor's checking.

We demonstrate a new class of rootkit exploits that modify specific return addresses on the stack. The new attack covertly executes by replacing return addresses from the `schedule` function for currently sleeping processes. Previous integrity monitoring systems could not prevent this attack because they check data structures by traversing them, pointer by pointer, from global variables. However, kernel stacks are large regions of untyped memory with potentially unspecified bounds and are thus resistant to traversal. Even software defenses for stack smashing attacks [2] cannot prevent this attack since such defenses detect overwrites to a region of stack space including a compiler inserted sentinel value. These defenses do not work when only the return address is overwritten.

The rest of this paper is organized as follows: §2 describes how OSck detects kernel rootkits. §4 describes our implementation of OSck as part of the KVM hypervisor. §5 describes kernel rootkits we used to evaluate OSck and the associated integrity properties these rootkits violate. We evaluate OSck in §6, discuss related work in §7, and conclude in §8.

2. Detecting kernel rootkits

Rootkits obscure signs of their presence, such as files in the filesystem or running processes, using a variety of techniques. There are three main techniques that rootkit writers use to hide the presence of a rootkit.

- Diverting control to code that hides presence of the rootkit.
- Executing code or storing data outside of a traditional process or file context.
- Modifying operating system data structures to conceal the presence of the rootkit.

A rootkit can divert control flow to modify queries about the state of the kernel, such as by hiding entries in the Linux `proc` filesystem by replacing pointers to functions implementing the filesystem with versions that conceal malicious running processes. It can also divert control flow so that the rootkit code runs outside of a context visible to the system administrator, such as running as a frequently invoked kernel callback rather than a traditional process. Additionally, rootkits can modify non-control data structures in a way that conceals its presence in the system. For example, a rootkit may remove a malicious process from kernel data structures used to list active processes in a system utility such as `ps`, but not from the data structures used to schedule processes.

2.1 Protecting control flow

A rootkit may modify operating system control flow to hide resources visible to user level monitoring tools. For instance, the adore-ng rootkit substitutes its own implementation of the Linux Virtual File System (VFS) `readdir` method for the `/proc` filesystem. The custom implementation hides certain entries from user queries about the contents of `/proc`, preventing a system administrator from viewing the full set of running processes. Alternately, a rootkit may modify control flow to run its code in a context undetectable by users. Our new "return-to-schedule" rootkit (§5), executes code in the context of every running process by modifying descheduled processes' stacks.

To detect rootkits that modify control flow, OSck verifies *control-flow integrity*, the property that execution of the kernel follows a predetermined control-flow graph [1]. We divide control transfers into three categories: *static*, *persistent*, and *dynamic*. Figure 1 illustrates these categories in a common system call.

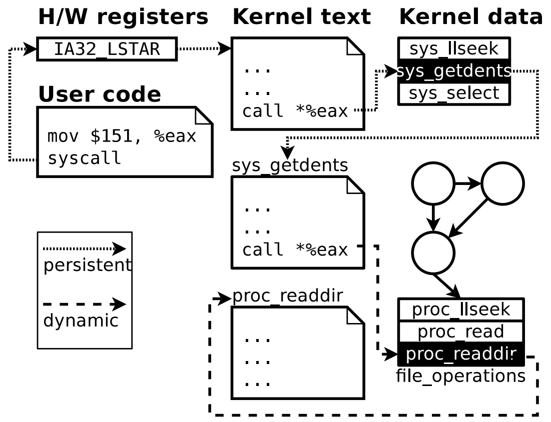

Figure 1. Control transfers in kernel invocation of a `getdents` system call. Static control transfers occur within blocks of kernel text. Persistent and dynamic control transfers use values in hardware registers and kernel memory to determine the next instruction executed.

Static and persistent control transfers *Static control transfers* consist of branches and function calls determined at kernel compile time, and are encoded in kernel text. *Persistent control transfers* are established by the kernel during initialization and remain constant while the kernel is executing, such as a system call invocation that is directed to the correct kernel text location via the values of hardware registers and the contents of the system call table (an in-memory table of function pointers). These register values and the contents of the table do not change once the kernel is initialized. OSck forces these transfers to remain immutable, by write-protecting kernel text, read-only data, and the values of special machine registers (§4.1).

Dynamic control transfers The kernel makes significant use of *dynamic control transfers*, which consist of indirect jumps and function calls based on pointers in writable kernel memory. The object-oriented design of the kernel encourages the use of function pointers to implement common interfaces to similar functionality. For example, virtual file system (VFS) modules associate tables of function pointers with kernel objects representing inodes and open files. To invoke the correct function for a given file, a generic kernel routine dereferences one of these pointers to perform a filesystem-specific function.

Unlike static and persistent control transfers, determining the valid target for a dynamic control transfer instruction in the running kernel may not be possible. For instance, two invocations of the same filesystem function on separate objects may result in different functions being invoked from the same call instruction in kernel text. The validity of the targeted function depends on a significant amount of context, such as the filesystem that contains the object and the type of filesystem object referenced. Instead of attempting to verify these complex correctness properties, OSck instead restricts rootkits' ability to execute arbitrary code by ensuring that dynamic control transfers will target functions that are *safe* for that particular callsite. Safe functions are part of known kernel or module text and share the desired function's *type signature*: the return type, and number and types of arguments.

The function pointer dereferenced in a dynamic control transfer is determined by a complex traversal of kernel data structures from some globally accessible root data structure. Because the exact traversal is only specified by the dynamic execution of kernel code, OSck must verify that all paths the kernel could traverse from

```
bool check_pidmap(){
  for_each_process(ptask){
    // bitmap entry for each process
    // should be set
    if(!test_bit(ptask->pid, pidmap))
      return false;
  }
  return true;
}
```

Figure 2. Simplified code that checks integrity of non-control data.

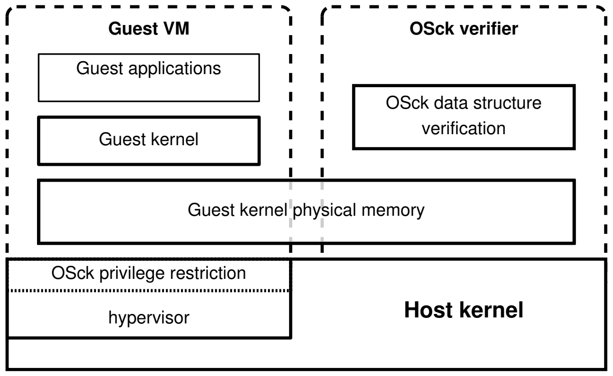
Guest VM
Guest applications
Guest kernel
OSck verifier
OSck data structure verification
Guest kernel physical memory
OSck privilege restriction
hypervisor
Host kernel

Figure 3. OSck architecture. Most of OSck's verification runs in a thread at the same privilege level as the guest OS, isolated by process and VM boundaries.

a global root to a function pointer will result in calling a safe function. OSck verifies this property by checking certain type-safety properties of the kernel heap (§3.2).

2.2 Protecting non-control data

In addition to modifying kernel control flow, a rootkit may also conceal system resources from the user by modifying non-control data in the kernel heap. For instance, rootkits based on direct kernel object modification [15] exploit the fact that kernels keep different data structures for enumerating and scheduling processes. In Linux, a rootkit can hide a process just by removing it from a hash table used for process enumeration. The hidden process is still scheduled normally.

Unlike type-safety of a graph of data structures, modifications to non-control data structures violate constraints that in many cases cannot be extracted from raw kernel source. In the example above, the invariant violated by the rootkit is that the hash table should contain the same set of processes as those in the process descriptor list and the run queue. Extracting such an invariant from source requires an understanding of code semantics that is beyond the scope of current and foreseeable tools.

Figure 2 is an example of code that checks the integrity of non-control data to protect the kernel from the `pidmap-hide` rootkit (§5) that hides itself from `/proc`.

For non-control data, OSck focuses on exporting a convenient interface for writing integrity checks. OSck extracts data structure definitions from kernel source and automatically generates an API for writing integrity checks as if they are code that would execute within a kernel thread, automatically handling translation of kernel addresses (§4.4).

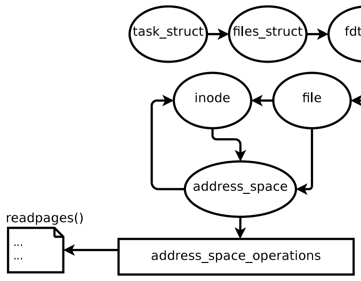

Figure 4. A subset of the kernel type graph. The kernel's traversal of this graph will result in the execution of the safe function `readpages`.

3. Design

OSck must verify the integrity of the running kernel, remaining safe even if the kernel is compromised. OSck runs as an independent thread at the same privilege level as the guest OS, but isolated from the guest by the hypervisor. Although it does not run as part of the hypervisor, the hypervisor must trust it (like domain 0 in Xen). Figure 3 summarizes the organization. OSck shares access to the guest OS' physical memory. Coordinating access to guest OS memory is the most hypervisor-specific part of OSck.

Many of OSck's integrity checks are based on verifying correctness properties of in-memory kernel data structures, so the most natural way to express integrity checks would be as code that could run in a kernel thread. Integrity checks should be allowed to reference kernel data structures using the same programming interface as kernel code. OSck extracts full type information from the kernel (§4) to support access to all kernel data structures (§4.3).

OSck detects rootkits by preventing or detecting modifications to static, persistent, and dynamic control transfers, and by verifying properties of non-control data. OSck protects static and persistent control transfers by restricting the guest kernel's ability to modify these values once they are installed. Dynamic control transfers are protected by verifying the type safety properties described in §2.1. OSck provides a programming environment identical to writing kernel code to make the job of verifying non-control data as easy as possible. Finally, OSck has the ability to quiesce the running kernel for any property whose verification fails due to the guest OS modifying data structures concurrently with OSck's checks.

3.1 Determining which code to trust

OSck detects malicious code in a running OS kernel by verifying that the hardware and memory values used to transfer control adhere to certain safety properties. For these checks to provide meaningful safety, OSck must ensure that both the code invoking control transfers and the code that the transfers target is code that is part of the original, non-malicious kernel implementation.

As a result, OSck must manage the regions of memory that are considered to be valid kernel code. We assume a trusted boot process via a technology such as TPM [31], to bring the system from a potentially untrusted kernel image on disk to executing a trusted kernel that may then be subject to OSck's verification. Modules dynamically loaded into the kernel must also become a part of the memory regions that OSck considers to be valid code. Currently, OSck relies on a whitelist and cryptographic hash of trusted modules. We believe that this solution is reasonable for a high-security setting. However, more robust solutions such as modules signed by a distributor are possible.

3.2 Verifying heap safety

Once the kernel is loaded in memory, it must not transfer control to malicious code while executing at the OS privilege level. OSck verifies this property by ensuring that any sequence of dereferences the kernel performs to reach a function pointer will result in a pointer to a safe function (defined in §2.1). OSck considers each node (a kernel data structure in memory) separately, verifying that the pointers contained in the data structure are consistent with the *type graph* specified by kernel code. A subset of the type graph for the Linux kernel is shown in Figure 4. OSck asserts that each pointer field within an object must point to the data type specified by that pointer, and all function pointers must point to safe functions.

Type information from an untrusted kernel OSck's type analysis maps arbitrary regions of kernel memory to kernel data types, often using kernel state that could be corrupted. A key challenge is to show that using type information from a possibly compromised kernel, it is possible to verify that any dereference path from a kernel root object to a function pointer must result in invoking a safe function. Here, we outline an argument that even a malicious kernel cannot evade detection by corrupting OSck's map of type information.

We consider two mappings of kernel addresses to data types, the *effective* mapping and the *bound* mapping. The effective mapping is the mapping implicitly created by the actions of kernel code. For a memory address, the effective mapping uses the type that kernel code implicitly assumes at that address, based on a sequence of pointer dereferences. That is, if the kernel traverses a `struct page` pointer to arrive at memory address 0x100, then memory at address 0x100 is of type `struct page`. We assume that the effective mapping is consistent with the kernel's type graph — the kernel does not typecast pointers to an unrelated type as it traverses them. A rootkit seeks to modify the effective map, changing kernel pointers such that the kernel believes a malicious function is legitimate, because a function pointer points to it.

The *bound* mapping is the map that OSck creates, starting with type information from the kernel source code, and augmented with potentially unreliable information on heap-allocated types. Memory locations in the kernel image (such as functions and static data structures) are bound to a type at link time, when a symbol name (with an associated type in the source code) is assigned an address. The type of memory locations on the heap are bound when dynamically allocated memory is scanned during runtime, through analysis of kernel memory data structures described in §4.2. The bound mapping and the effective mapping will agree for all global roots. Global roots are symbols compiled into the kernel, and they are protected by OSck by write-protecting the kernel text that uses those symbols as constant values. In addition, the bound mapping maps the entry points of all safe functions to their type signature. For both global roots and safe functions, a compromised kernel cannot change the bound mapping, since OSck does not rely on dynamic kernel state for those addresses.

In the presence of a rootkit, the bound mapping must differ from the effective mapping in at least one place: the malicious function. The kernel believes the memory at the malicious function is a valid function, while OSck, which binds the addresses of safe functions when the kernel is linked, and does not rely on the values of mutable function pointers, will not.

If a rootkit inserts malicious functionality into the kernel by modifying a function pointer, there is a traversal of the data structures in the kernel's effective map that begins at a global root and ends at the malicious function. At the global root, the effective map and bound map must be consistent, since OSck protects the global roots. At the malicious function, the effective and bound maps must be inconsistent. Thus, there must be a step in the traversal, a pointer in object A to a destination location B, for which the maps have A mapped consistently, and B mapped inconsistently with each other. We assume that the effective map is consistent with the type graph. Then for OSck, which uses the bound map, the pointer from A to B must be inconsistent with the type graph. If it were consistent, then B would be mapped to the same type as in the effective map. This pointer will fail OSck's verification process, and OSck will detect the presence of a rootkit.

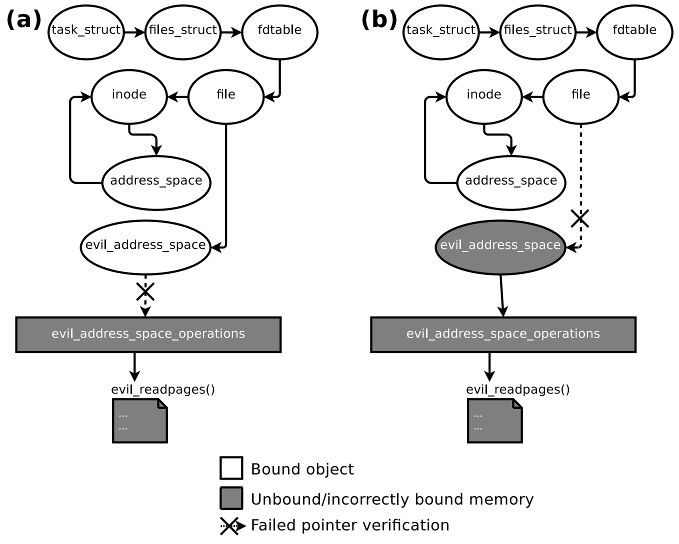

Figure 5. Modified type graph created by a malicious modification to the kernel heap, with two possible bindings from object to type. Regardless of binding, malicious modification will cause verification to fail.

Type-safety example Consider a typical Linux file read that traverses the data structure graph from a `file` object to the associated `address_space` to the table of `address_space_operations`, and finally to the `readpages` method. The type graph on which this traversal operates is illustrated in Figure 4. A kernel rootkit may divert this traversal in several ways that result in the kernel invoking an unsafe function. One possible modification is illustrated in Figure 5. A rootkit may allocate a duplicate `evil_address_space` that points to a duplicate `evil_address_space_operations`, which contains a pointer to the rootkit's custom code. If OSck binds the `evil_address_space` to the `address_space` type, as in Figure 5(a), but the `evil_address_space_operations` remains unbound or is bound (based on incorrect information from the kernel) to some other type, verification will fail because the pointer to the `evil_address_space_operations` is not type-safe. If neither object is bound, as in Figure 5(b), verification will fail because the pointer from the `file` object to the `evil_address_space` is not type-safe. OSck uses kernel memory management data structures to bind kernel memory to data types without requiring developer interaction. §4.2 describes this process in detail.

OSck verifies the entire heap in a linear scan of kernel memory. This approach has several advantages. First, a linear scan is efficient: in the common case, checking pointer targets is a read-only operation. It does not require marking objects as visited, updating queues of unvisited objects, or making recursive calls. In our benchmarks, checking type-safety for much of the kernel heap requires at most around 300ms, and as little as 50ms (§6.2). Second, our approach facilitates incremental verification. A performance-sensitive system, or a system unable to concurrently run kernel and verification code may require that the heap be verified in stages. OSck's linear scan facilitates easy recording of the position in the heap to be resumed later.

3.3 Handling concurrency

A key goal of OSck is to minimize the performance impact of monitoring kernel state by interleaving execution of OSck's verification process and the kernel's execution as much as possible. However, running the kernel verifier concurrently with the kernel presents several synchronization issues. In general, OSck's verification process may attempt to check kernel state that the kernel is in the process of updating by means of a code region that it believes to be atomic. For example, OSck could read a pointer from a structure that is incompletely initialized and whose contents could point anywhere, or it could observe a red-black tree in the process of rebalancing and conclude that an integrity constraint has been violated.

Although OSck's access to kernel memory could allow it to correctly synchronize with the running kernel, doing so would significantly increase complexity and reduce performance. OSck frequently scans large portions of the kernel heap. Correct synchronization would require acquiring and holding a large number of locks, and adhering to the locking discipline of the existing kernel. In addition, there are synchronization requirements in the kernel that can depend on the order in which data structures are accessed, adding additional undue complexity and requiring OSck to follow numerous rules specified only through comments and a particular kernel implementation. OSck is designed to minimize effort when porting to successive kernel versions. Incorporating details of kernel synchronization into OSck would require additional expertise to verify that OSck's synchronization conventions accurately track the kernel's.

Instead, we make two assumptions that allow OSck to verify the state of the kernel with no synchronization in the common case. First, we assume that false negatives due to unsynchronized accesses are transient. OSck detects persistent modifications to kernel state, and it is unlikely that malicious code could consistently modify kernel state to be correct when observed by OSck, but incorrect when viewed by the kernel. Second, we assume that false positives due to unsynchronized accesses are rare. OSck may then employ a heavyweight synchronization mechanism in the event of an integrity violation to bring the system to consistent state and re-verify. In the common case that the event reflects an actual integrity violation, performance is not critical. In the rare case of a false positive, that heavyweight synchronization will have negligible performance impact. This assumption is validated by our experiments. While we are able to construct microbenchmarks in which false positives are frequent, we did not observe any false positives in any of the benchmarks that we ran (§6.4).

In the event of an integrity violation, OSck performs two steps before alerting the user about the possible rootkit infection:

- Immediately re-verify the state of the kernel. Kernel atomic regions are short, so re-running the check should resolve many integrity violations due to concurrent accesses.
- Enforce *quiescence*. OSck halts the kernel at a point at which all data structures should be consistent, and then re-verifies kernel state.

Choosing an effective point at which to enforce quiescence is an important part of OSck's design. An effective quiescence point should have two key properties. First, it should be a point where kernel data structures are consistent. Second, a multi-processor system requires enforcing quiescence for each processor. A processor should not quiesce in a place where it holds resources required by another processor (such as a lock), thus preventing quiescence for the second processor and creating deadlock. Our implementation is discussed in §4.5.

3.4 Discussion

As with other systems that attempt to detect security violations by inspecting the state of a potentially insecure system (e.g. a virus scanner), it may be possible for a sufficiently advanced rootkit to evade detection by OSck. OSck currently detects the attack vectors used by all of the Linux rootkits that we were able to locate. In this section, we discuss our assumptions, and ways in which a sufficiently advanced rootkit might evade detection.

Threat model The threat model we consider for OSck is that of an attacker with temporary read-write access to kernel memory. For example, an attacker might gain access to kernel memory through a bug in the kernel, or by exploiting a setuid program and accessing the (now-deprecated) `/dev/kmem` interface. An attacker with access to kernel memory could corrupt kernel state and cause the kernel to crash. OSck does not protect against these modifications, or similar attempts to deny service. Instead, the goal of OSck is to detect *meaningful* malicious modifications to kernel state: modifications that facilitate further malicious activity in a way that will not be detected by a system administrator. For instance, an attacker could spawn a daemon as the root user, and then use modifications to kernel memory to hide that daemon from the list of running processes. The attacker must modify kernel memory carefully enough that queries into system state return a reasonable set of results while hiding some set of information.

Type-safety A majority of kernel rootkits (see §5) implement malicious behavior by subverting kernel control flow. Many control transfers within the kernel are relatively simple to protect by protecting modifications to low-level state, such as the contents of kernel code and architectural registers.

The kernel makes extensive use of function pointers to transfer control, thus OSck must also protect the values of function pointers in writable kernel memory. OSck protects function pointers by ensuring that any function pointer used by the kernel points to a function that is a part of already trusted code, and has the correct type signature. This policy prevents all current attacks on function pointers, which direct pointers to custom rootkit functionality.

An attacker that wishes to remain undetected may only swap pointers with other functions that share the same type signature. We believe that implementing meaningful malicious behavior (such as hiding a single entry in the `/proc` filesystem) under these constraints is unlikely to be successful.

Kernel data structures In addition to subverting kernel control flow, an attacker can hide evidence of malicious activity by subverting implicit invariants between kernel data structures, for example in the Linux kernel the assumed equivalence between the tree used to schedule processes and the linked list used to display them. OSck protects these data structure relationships through ad-hoc checks on kernel state.

For a large production kernel such as Linux, specifying the complete set of invariants is not likely to be tractable. However, to provide effective security, OSck need only check invariants that rootkits can violate to implement meaningful malicious behavior. In addition, adding a new integrity check to OSck's library is significantly less intrusive than fixing a kernel exploit, and does not require restarting a system. Checks in OSck can be implemented in a handful of lines of code, and do not need to be as thoroughly tested as patches to a production kernel: a crash in OSck requires only that it be restarted, without affecting the running kernel.

Concurrency OSck's checking code runs concurrently, and in the common case unsynchronized, with kernel code. We make

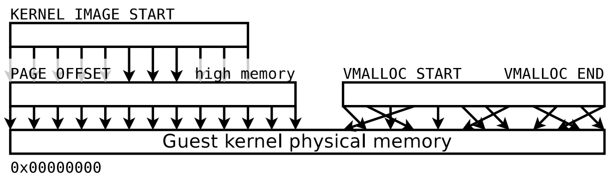

Figure 6. Mapping guest OS memory into the OSck verifier address space.

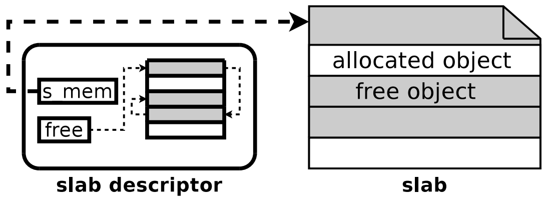

Figure 7. Slab organization of kernel memory allocations. Information about allocated and free objects within a slab is contained in a slab descriptor's internal list.

two main assumptions regarding concurrency. For performance, we assume false positives (detecting a violation of some invariant due to concurrent modification and not malicious behavior) are rare. Our experiments (§6) support this assumption. For security, we assume that false negatives are rare and not reproducible by an attacker. We recognize that race conditions form a significant body of attack vectors on a wide variety of systems. However we note that in general a race must happen once to violate security, while in OSck an attacker would have to win every race for the lifetime of the system to evade detection. We also assume that an attacker cannot accurately predict when an OSck check will run, so attack code cannot repair kernel data structures just in time for the OSck check, and then corrupt them again after the check.

4. Implementation

We implement OSck as part of the KVM hypervisor [20]. KVM virtualizes a guest kernel within a host Linux kernel. A user-level *launcher* binary allocates a region of memory to serve as the guest's physical memory, loads the guest kernel, and then calls into the host kernel to run the virtualized guest. OSck runs as a separate process, communicating over a socket with the launcher binary. OSck reads guest state by mapping the same memory region as the launcher.

OSck integrity checks are written as if they were executed in a guest OS kernel thread, although they are isolated from the guest kernel by the hypervisor. This is possible by mapping several virtually contiguous memory segments from the guest kernel into the OSck verifier address space, as shown in Figure 6. The majority of kernel data structures are addressed in a large portion of the virtual address space that linearly maps physical memory. In the 32-bit x86 Linux kernel, this consists of the last gigabyte of a 4GB address space. On the 64-bit kernel, a 64TB region starting from `0xFFFF880000000000` maps physical memory. A second region (the same region in 32-bit Linux) maps text and data in the kernel image.

In addition to the linearly mapped region, several kernel data structures (such as the list of kernel modules) reside in a region of memory allocated by the `vmalloc` function, which may contain non-linear mappings of physical memory. The OSck verifier translates addresses in this area by reading guest page tables from physical memory. Although reading page tables in software is inefficient, few kernel data structures reside in this area, and so this does not noticeably increase OSck's overhead. These memory mappings give the OSck verifier the same view of kernel data structures as a kernel thread. Our original OSck prototype was developed for 32-bit Linux. Porting OSck to a 64-bit architecture required few changes other than redefining a few aspects of the kernel's memory layout and interpreting a different page table format.

Both automatic and manual integrity checks in OSck make significant use of data structure definitions and properties extracted from kernel source code. As part of kernel compilation, OSck translates the kernel into the C intermediate language (CIL)[22]. We use CIL to output sanitized kernel type definitions for use in manual checks and to automatically generate functions to verify type-safety for each data type. The executed kernel is compiled directly using `gcc`; the CIL processing step is only to generate code that executes as part of OSck integrity checks.

This section explains how OSck protects static and persistent control transfers by restricting guest privilege (§4.1). Dynamic control transfers are protected by verifying type-safety properties (§2.1). Verifying these properties requires a mapping from locations in kernel memory to the data structure type. §4.2 describes how we use existing kernel memory management structures to efficiently construct a mapping. OSck provides a convenient interface to kernel data structures that is used both by automatic verification code (§4.3) and system programmers specifying custom integrity properties (§4.4). §4.5 discusses the synchronization protocol between the guest kernel and OSck.

4.1 Restricting guest privilege

A query about the state of the system, in the form of a system call, begins with a user executing the `syscall` instruction with a system call number specified in the x86 `eax` register, or an `int 0x80` instruction with the values contained in the x86 interrupt descriptor table. Either way, the path of control from user-level to the kernel implementation of the system call is determined by the value of hardware registers, kernel data, and kernel text. These control transfers remain fixed once they are installed by the booting kernel.

OSck protects the static and persistent control transfers on system call entry by restricting the guest operating system's privilege to modify certain registers and regions of memory once their values are set during boot. Modifying hardware registers that determine the transfer of control flow from user to kernel mode (such as x86 machine-specific registers that determine the operation of the `syscall` instruction) is a privileged operation, and must be implemented through a call to the hypervisor. OSck modifies the hypervisor to prevent updates to these registers after the initial values are installed.

After a user system call traps into the kernel text segment, code transfers control to the service routine corresponding to the desired system call number by indexing the *system call table*, an array of function pointers in kernel memory. The kernel does not modify the system call table once it has been created. OSck protects the system call table through hardware page protection and a hypervisor call ensuring that once the table is initialized the guest operating system may not modify it. Similarly, the implementations of kernel functions, such as the routines that implement individual system calls, remain static once the kernel is running. As with the system call table, OSck freezes read-only hardware page protections on pages containing kernel text, preventing the guest kernel from modifying their contents.

4.2 Slabs for efficient type safety

Efficiently constructing an accurate mapping between kernel memory and data type is key to both OSck's integrity guarantees and performance. Most important data structures in the Linux kernel are allocated via *slab allocation*, which provides *caches*, or specialized per-type allocators [7]. Kernel caches allocate memory from groups of pages called *slabs* containing a single type of object. Important kernel data structures that are dynamically allocated from caches include directory entry (`dentry`) structures used by kernel filesystem lookup routines, `buffer_head` structures that describe on-disk locations of blocks that make up a memory page, and `radix_tree_node` structures that comprise radix trees used, for example, in indexing pages in page caches.

OSck can determine the type of memory allocated within each page by determining the type allocated by a kernel cache. The Linux kernel maintains data structures that allow a given memory address to be associated with metadata about its containing page. For pages that are part of a slab, the metadata contains a pointer to the *slab descriptor*. The slab descriptor, shown in Figure 7 specifies the cache that contains the given slab, alignment information for objects within the slab, and a list of free objects.

OSck uses this kernel-provided information to perform efficient mapping between memory and data type. OSck associates each cache with a data type, and binds every allocated object in the slab to that data type. Objects that are not allocated or not part of a slab are unbound.

OSck associates individual kernel caches with data types by analyzing kernel source code during compilation of the kernel binary. The addresses of most kernel caches are stored and accessed through global variables. To allocate an object from a kernel cache, a cache variable is passed to a generic allocation function, and the resulting untyped pointer is cast to the desired type. OSck's analysis of kernel source code detects these sequences and associates the cache variable with the type that it allocates.

4.3 Automatic type-safety checks

The wealth of data types within the kernel requires that OSck automatically generate per-type checks for verifying the type safety of the kernel heap. For each data type, OSck generates a C function that takes a pointer to kernel memory as input. For each pointer within the data type, OSck outputs a call to our pointer-based type safety verifier. For each nested data type, OSck outputs a call to the function responsible for verifying that data type.

Most kernel type definitions are unambiguous. However, some types include a C `union` to pack multiple fields of information within a single region of memory. Without higher-level information, OSck is unable to disambiguate the different fields of a union to call the correct verifying function. Instead, OSck considers a union correct if at least one of its fields verifies correctly.

4.4 Manual integrity checks

Although many rootkits violate kernel control flow in a way that may be detected by automatically generated type-safety verification, there are many integrity properties important to kernel operation that cannot be automatically extracted from kernel code. For instance, the relationship between the bitmap of allocated process ids and the list of running tasks is unlikely to be extracted from kernel source code. For these properties, OSck exports an interface for writing verification code that is convenient for those who understand kernel internals best: kernel developers. We believe that hand-written code for checking data structures should resemble as closely as possible code that runs within a kernel-level thread.

There are two challenges to allowing developers to write natural verification code within a user-space program. First, the wide variety of data structures within the kernel must be made available, without the complication of including kernel header files in user code. Second, authors of integrity checks should be able to seamlessly use pointers within kernel data structures, without considering the translation necessary to reflect the kernel address space within a user process.

As part of its analysis phase, OSck automatically generates a convenient API for writing kernel integrity checks. OSck collects kernel type definitions and performs a source to source translation, generating a header file containing type definitions with memory layouts identical to those in the kernel, without preprocessor and other complexities common to kernel header files. OSck sanitizes identifiers for safe usage within C++, and transforms all pointers within data structures to smart pointers. These pointers automatically translate from kernel addresses to addresses in the user-level reflection of the kernel address space, illustrated in Figure 6. Pointers within the linearly-mapped region of kernel memory translate directly to the launcher process' map of guest physical memory. Pointers within the `vmalloc` region translate to the launcher process' map of the kernel virtual address space.

We have found OSck's automatically generated kernel data structure API useful in practice. Kernel convenience functions and macros, such as for iterating through kernel lists, may be copied without modification and perform correctly on a more complex translation of the kernel address space. Nearly all of the verification code in OSck is written using this API, including the large amount of code dedicated to caching kernel type information and verifying type-safety for kernel pointers.

4.5 Quiescence

To minimize checking overhead, OSck verifies kernel data structure integrity in a thread that runs concurrently with guest execution. In the common case, when OSck does not synchronize with the guest kernel, data races are possible. OSck must manage errors and false positives due to races efficiently and safely. First, OSck maximizes performance when data structures are not being updated, or inconsistencies due to races are transient. Second, OSck provides a means for the hypervisor to synchronize with the guest to check properties that are violated because of guest activity.

It is possible for the guest OS's workload to prevent an integrity check from succeeding. For example, we present an experiment in Section 6.4 where several user programs are constantly yielding the CPU, making it difficult for the OSck hypervisor to verify that the rbtree representation of the run queue contains the same processes as the list representation (the kernel assumes the same processes are in both representations). To check this property, the hypervisor must quiesce the system, which suspends execution of the virtual processors of the guest OS.

As discussed in §3.3, OSck attempts to verify data structures and when it fails, it quickly re-verifies (to see if the problem was transient and has disappeared). If the re-verification fails, it enforces *quiescence*.

The guest's virtual processors must be suspended while kernel data structures are consistent, otherwise it will look to the hypervisor like a transient problem is persistent. There are locations in the guest where all guest data structures are consistent, and these are potential points at which the hypervisor can suspend virtual processors. However, it is also important only to select points such that all kernel threads can reach quiescence. If a guest kernel thread on a suspended virtual processor held locks that were necessary for another kernel thread to reach a quiescence point, then attempting to induce quiescence would instead induce deadlock.

OSck quiesces the kernel at the start of the `schedule` function. The function is called frequently, minimizing the response latency of the guest when the hypervisor requires quiescence. Although some kernel locks may be held when `schedule` is called, all such

Type	Name	Data structures exploited
Control flow-related	ret-to-sched*	Return address of suspended kernel thread's stack
	extable*	Exception table of the inserted module
	adore-ng%	`file_operations` table of `/proc`
	enyelkm%	Kernel text and page table
	dbg-reg	Debug registers and the `notifier` chain
	net3	Protocol handlers and netfilter filters
	proc-hide	`file_operations` table of `/proc`
Non-control	hideme†	Thread list for a process id
	pidmap-hide†	Bitmap for PID allocation

Table 1. Summary of rootkits used in our evaluation. Asterisks (*) indicate new classes of rootkits proposed in this paper. Percent signs (%) indicate actual rootkit code, ported to the latest 2.6 kernel. Daggers (†) indicate new implementations of known attack techniques.

locks are *blocking*. If another kernel thread attempts to acquire a lock held by a virtual processor that has quiesced in the `schedule` function, it will yield the processor by calling `schedule` itself. Instead of deadlocking, the processor attempting to acquire the lock will similarly quiesce.

OSck enforces quiescence in two stages. First, it interrupts the guest kernel to halt its execution. OSck overwrites the beginning of `schedule` with a hypercall after ensuring no guest thread is in the midst of executing the replaced instructions, and allows the guest to resume. When the guest attempts to call `schedule`, the hypercall transfers control to OSck. OSck verifies the entire state of the quiesced kernel, restores the original instructions in `schedule`, and resumes execution at the beginning of the function. The guest the continues as if it had completed a simple call to `schedule`.

5. Rootkits

We evaluate OSck with several publicly-available Linux rootkits, that corrupt the integrity of a running kernel via several vectors. The source code of many of these are specific to the 2.4 version of the kernel. We ported existing rootkits to work with the latest 2.6 kernel, sometimes specializing them to separate out different attack vectors.

In addition, we present several kernel exploits that to the best of our knowledge are not currently used by rootkits. These are Linux exploits developed out of our familiarity with the kernel. While having the source code of Linux was valuable in writing these exploits, we believe that programmers who work with the APIs of a closed source operating system like Windows would be able to create similarly creative exploits. Table 1 summarizes the rootkits used in this study.

Return to schedule The ret-to-sched rootkit overwrites the return address on the stack of every process blocked for I/O. Rather than return to the `schedule` function, processes return to the rootkit code, which then returns to `schedule`. This exploit is undetectable by all previous integrity checking systems we are aware of. It does not modify any kernel data structure reachable from pointers in the kernel heap. By modifying a single stack location, there is no "collateral damage" (overwrites of compiler-placed sentinel values) that techniques like StackGuard [10] rely on to detect a stack smashing attack.

The ret-to-sched rootkit runs in the context of many processes by rescanning the I/O queues on each invocation and reinfecting any processes that are blocked. The attack depends on the rootkit understanding the stack layout of the compiler, though in the case of Linux there are few compilers besides `gcc` that can compile a functioning kernel.

Because ret-to-sched does not create or perceptibly alter the number or state of running processes, it is not detectable by any system administration utilities. Listing the system's active processes will not detect anything out of the ordinary. OSck detects this attack by ensuring that the return addresses on the stacks of descheduled processes correspond with valid kernel code regions. Stacks are not currently checked to determine if they represent a valid call stack, so a sufficiently advanced attacker could construct a return-oriented program [28]. However, OSck could be extended to add this property, by including information on the valid kernel call graph in its analysis.

extable The extable rootkit uses the kernel's exception handling mechanism to subvert control flow. The exception table is a necessary kernel feature that speeds up the common case of copying data to and from the user. On every page fault, the Linux page fault handler searches the exception table of the kernel and loaded modules using the faulting PC as the key. If a match is found, control is transferred to the value in the exception table. The extable rootkit inserts an entry that associates a faulting address in the kernel text section with a fixup address within the module text section (or dynamically allocated kernel memory). Any page fault at the specified kernel PC will cause execution flow to resume at the fixup PC that is specified by the module.

The rootkit corrupts a kernel global variable that is dereferenced in a single location in kernel code, and is never copied into another data structure. The variable is dereferenced whenever a workqueue (a Linux mechanism used to defer work in interrupt handlers) is created. On this common event, the kernel faults, the rookit executes and finishes by patching up the machine registers to make it appear as if the dereference succeeded. The pointer remains corrupt, so the rootkit can be called again.

adore-ng adore-ng modifies structures that contain sets of function pointers used by Linux's Virtual File System (VFS) layer to accommodate many different filesystems within one uniform directory hierarchy. adore-ng has a user-controllable security password, and a lookup of `<password>-fullprivs` as a filename within `/proc` will call adore-ng's `lookup` function, giving the task performing the lookup root privileges. This is used as a method of obtaining root privileges on subsequent login after an initial intrusion.

The Linux developers have taken steps to make this exploit more difficult. Beginning in kernel version 2.6.16, data marked `const` is placed on write-protected pages, and all data structures containing these function pointers were marked `const`. Writes to these structures are no longer possible without modifying permissions in page tables. However, when used through VFS, the operations are actually referenced via pointers from directory lookup cache entries (`dentry`). We modified adore-ng to change the pointers within `dentry` structures to point to rootkit allocated VFS function pointer structures with malicious function pointers. This example indicates how difficult it is to defeat rootkits from within the kernel. OSck ensures the integrity of individual function pointers, and will detect the malicious pointers in adore-ng.

enyelkm enyelkm hides files and processes by modifying code that dispatches to the system call table. The system call code is patched to redirect system calls to modified versions. A new version of `getdents` conceals directory entries, such as those in the `/proc` filesystem used to enumerate running processes. In addition, enyelkm modifies the `read` system call to not return portions from files that are contained within specially delimited sections.

As a result, enyelkm can survive reboot by allowing an adversary to modify startup configuration files in a way such that the enyelkm module loads on boot, while the configuration file changes will then be subsequently invisible due to the modified `read` system call. Detectors that examine the system call table contents themselves will not detect this rootkit.

In order to be able to modify the system call dispatch code on recent version of Linux, we modified enyelkm to change the page table permissions for the system call code region. OSck detects this modification and flags it for violating the integrity of the system call table.

dbg-reg dbg-reg is a specialized version of the mood_nt rootkit that uses x86 debugging registers and breakpoint notifiers to alter kernel control flow.

The x86 has four debug registers that can hold addresses which, when executed, cause breakpoint exceptions. Linux handles breakpoint exceptions using a chain of callbacks, named notifiers. On a breakpoint exception, the notifiers on the chain are called one by one until the end of the chain is reached, or one of the callbacks returns a special return code.

dbg-reg registers a notifier and puts the address of the `call` instruction that dispatches function pointers from the system call table into a debug register. The rootkit dispatches from a modified system call table that contains pointers to malicious functions. The control register (DR7) has a protection bit which helps disguise the rootkit. When the general detect enable bit is set, any attempt to read or write the debug registers causes a debug exception, which transfers control to the dbg-reg handler. In this case, the handler emulates the read or write and returns a code preventing other handlers from running. Code that reads or writes the registers believes it has succeeded. OSck defeats this rootkit by disallowing the guest OS to set debug registers with kernel addresses. In a production setting where security is paramount, debugging support for the kernel is not necessary.

net3 net3 implements an attack [26] where protocol handlers and netfilter filters take arbitrary action upon receiving network traffic (specifically, ICMP traffic). The rootkit netfilters pass the ICMP traffic, but can be configured to drop incoming or outgoing traffic that matches arbitrary criteria. Dropping incoming traffic can disguise input used to signal the rootkit. Dropping outgoing traffic can disrupt a component of a distributed application executing on the machine.

This rootkit can be stopped by ensuring that functions called by protocol handlers and netfilter filters are in legitimate address regions—either within kernel text or within the text of a set of allowed modules. OSck is notified when the kernel loads modules, so it verifies that the netfilters function pointers are legitimate.

Non-control flow based rootkits hideme and pidmap-hide directly manipulate the relationship among kernel data structures instead of altering control flow. Both rootkits hide a process from the `/proc` directory. When the user lists the contents of the `/proc` directory, the kernel enumerates all processes, creating a directory for each PID. hideme removes the target process from the `pid_hash`, hiding it from the PID enumeration.

Because PID use is sparse, the kernel first checks the `pid_hash` table when enumerating PIDs, and then uses the `pidmap` bitmap to find the next in-use PID. pidmap-hide clears the bit in `pidmap` that corresponds to the target process. But the kernel looks in `pid_hash` first without looking at `pidmap`. To hide PID p, the rootkit must prevent any running process from having PID $p-1$, which it does by running several processes in quick succession to get two with adjacent PIDs. The cleared bit in `pidmap` will be detected and corrected by the kernel when PID allocation overflows. The rootkit delays PID overflow by setting the last allocated PID to $p+1$ where p is the PID for the process hidden by the rootkit.

Rootkit	OSck action	Detection time (s)
ret-to-sched	Detected	0.51
extable	Detected	0.51
adore-ng	Detected	0.50
enyelkm-1.3	Prevented	-
dbg-reg	Prevented	-
net3 (protocol handler)	Detected	0.44
net3 (netfilter filter)	Detected	0.48
proc-hide	Detected	0.46
hideme	Detected	0.56
pidmap-hide	Detected	0.74

Table 2. Rootkits detected by OSck

Integrity check	LOC	Rootkits
Guest privilege restriction		enyelkm, dbg-reg
Heap type-safety	504	adore-ng, proc-hide
ck_extable	64	extable
ck_net_filter	74	net3
ck_prot_handlers	64	net3
ck_proc_namespace	99	hideme, pidmap-hide
ck_stacks	40	ret-to-sched

Table 3. Kernel integrity checks implemented by OSck. Lines of code for each check are reported as calculated by SLOCCount [33].

6. Evaluation

In this section we measure the performance and detection capability of our OSck prototype implemented in the KVM hypervisor. All experiments were performed on a Intel 2.80GHz Core i7 860 CPU, with 8GB memory and running the 64-bit Linux 2.6.32.9 kernel. We ran virtualized guests with 2GB memory, and in 1- and 4-core configurations.

6.1 Rootkit detection

Table 2 shows the set of rootkits we created or ported, and the average time OSck takes to detect them. Rootkits listed as "Detected" were detected by OSck's periodic integrity checks. Rootkits listed as "Prevented" were prevented from modifying the kernel by OSck's restriction of guest privilege. We ran each rootkit a minimum of 5 times to make sure OSck detected it each time, and report the average detection latency for those detected by integrity checks. Our experiments show that OSck detects rootkits reliably and promptly. We observed no false positives: every non-rootkit benchmark run with OSck did not report a rootkit.

Table 3 lists OSck's integrity checks. Heap type-safety requires about 500 lines of code, but all other checkers were less than 100 lines. The largest checker, `ck_proc_namespace`, verifies two independent properties of the `/proc` filesystem, and detects two rootkits. In general, integrity violations caused by a wide class of kernel rootkits are defeated with very small checking functions.

6.2 Performance of OSck integrity checks

To illustrate the low overhead of OSck's integrity checks, we measured the time necessary to complete an iteration of the full set of checking code, including verifying type-safety for much of the kernel heap (Table 4). The amount of time to check the integrity of kernel data structures varies by the number of data structures a given benchmark creates on the heap. Our kernel compile benchmark requires on average 126ms to complete a complete iteration of OSck's checking code. RAB, which can create 10 times as many

Benchmark	Avg. check time (ms)	Max. check time (ms)
SPEC INT	76	123
SPEC FP	43	109
RAB	109	316
Kernel compile	126	324

Table 4. Execution times for a single iterations of OSck's complete integrity checks for each single core benchmark.

Single core			
	host	guest	OSck
SPEC 2006			
INT	1.00	1.03	1.05 (2%)
FP	1.00	1.03	1.03 (0%)
PARSEC			
raytrace	13.2	13.4	13.5 (1%)
swaptions	11.5	11.5	11.5 (0%)
x264	2.76	2.87	2.87 (0%)
RAB			
mkdir	5.62	5.87	5.98 (2%)
copy	17.68	44.07	45.15 (2%)
du	0.29	0.39	0.40 (3%)
grep/sum	2.51	1.89	1.86 (-2%)
Kernel compile			
2.6.32.9	515	471	473 (0%)

Multiple core			
	host	guest	OSck
PARSEC			
raytrace	7.87	8.05	8.17 (2%)
swaptions	3.43	4.24	4.11 (-3%)
x264	0.97	1.20	1.17 (-3%)
Kernel compile			
2.6.32.9	145	179	189 (6%)

Table 5. Benchmark execution time (slowdown of OSck checks over guest execution in parentheses). SPEC reports the weighted geometric mean of slowdown, the remainder report seconds.

objects as an idle system, requires an average of 110ms over the entire benchmark. We report the average and maximum times for any iteration of OSck's checks to describe the range of the latency.

For a given checking latency, a user can set the checking interval to achieve a particular performance level. For example, if OSck checks take about 50ms, then running them every 5s will slow down the workload by 1%. Rootkit detection on the order of seconds is the goal of OSck. If a multiprocessor host has a free CPU, OSck checks can mostly be overlapped with guest computation. Quiescence is rare in our benchmarks (§6.4), so its latency can be safely ignored.

6.3 Benchmarks

Table 5 compares the execution time of several benchmarks on the host machine, executing within the KVM hypervisor, and executing on KVM with OSck integrity checks run every 0.5 seconds. All results are the average of 5 runs.

We first describe the slowdown for several compute-intensive workloads. For SPEC 2006 [14], we report the weighted geometric mean for the slowdown of integer and floating point workloads. The results show that OSck has less than a 2% or lower performance overhead over KVM at this checking interval.

We ran a subset of the PARSEC [6] benchmark suite to verify that OSck continues to minimally impact performance in a multi-core setting. We report the average of 5 runs for three different applications: a ray-tracer, a securities market simulator, and an H.264 encoder. We used a four core VM for this benchmark. OSck exhibited a maximum of 2% overhead in any one application.

As an I/O bound benchmark, we ran the reimplemented andrew benchmark (RAB) [24], a scalable version of the Andrew benchmark [16] that uses several phases to stress the file system. In our configuration RAB initially creates 100 files of 1024B each and measures the time for the following four operations: (1) creation of 20,000 directories, (2) copying each of the 100 initial files to 500 directories, (3) executing the `du` command to calculate disk usage, and (4) searching for a short string and checksumming all the files. Our experiments show that OSck imposes a maximum overhead of 3% compared to the guest OS running in the unmodified KVM. This workload has over an order of magnitude more kernel objects than the other workloads.

Finally, we report the performance of a kernel compile, a workload that balances I/O and CPU demands. In the single core case, execution time does not change compared with unmodified KVM, and for four cores, the overhead is 6%.

Num procs	Failed checks
5	14%
10	19%
50	23%

Table 6. Percent of OSck's runqueue integrity checks that fail for different numbers of user processes that do nothing but yield the processor.

6.4 OSck and concurrency

A key goal of OSck is behaving correctly in the face of concurrency. Previous systems have assumed that any false positives when monitoring the kernel heap due to data races would be transient. However, we present a user-level benchmark that consistently violates an integrity property due to concurrency. One technique for hiding malicious processes is to create inconsistency between structures used for scheduling processes, and those used for listing processes. We run an integrity checker that compares the list and rbtree representations of the scheduler runqueue, as several user-level programs continuously yield the CPU. This creates a steady stream of updates to the kernel's runqueue data structures. Because OSck checks the integrity of these data structures while they are being modified, it can see an inconsistent view. Table 6 shows that as the number of user programs increases, the likelihood of seeing an inconsistent view rises. Many previous integrity checking systems simply reverify a failed check a fixed number of times (e.g., 3), and this data shows that such a strategy is not sufficient. Even with 5 processes, 14% of checks find an inconsistent view making a false positive from 3 consecutive inconsistent views an eventual certainty.

OSck only quiesces if two consecutive checks fail, so a mechanism to deal with repeated failures is necessary in practice, though rare enough that a heavy-weight mechanism is acceptable. There were no false positives—all checks passed after quiescence in this concurrency stress-test. We found no occurrences of quiescence in our other non-rootkit benchmarks.

7. Related work

Since the time that VMM-based introspection was originally proposed [13], there has been significant progress on techniques for monitoring kernel integrity from a hypervisor. OSck integrates many of these techniques into a single practical package that stresses cooperation between the hypervisor and the monitored kernel. Just as paravirtualization [5] modifies kernel device drivers to make virtualization more efficient, OSck integrates type information and an understanding of the kernel memory allocators to make integrity checking comprehensive and efficient. OSck requires only minimal modification to the guest kernel, generally to communicate boot time parameters like when to begin enforcing immutability of protection on kernel text page table entries.

VMM-based approaches to security can be categorized by the types of guarantees that they provide for a guest VM. Systems like Terra [12] attest, via a chain of trust, that the VMs are initialized to a desired state. Terra [12] guarantees that its virtual machines cannot be inspected or modified by other VMs. Attested VM initialization and its counterparts like secure boot [3, 19] are orthogonal to the design goals of OSck. OSck is instead concerned with the continued maintenance of guest OS integrity after initialization, and Terra provides no such guarantee.

After correct initialization, the next guarantee is that of guest kernel code integrity. SecVisor [27] protects kernel code from modification thus defeating certain attacks. However, merely protecting code segments is insufficient to guarantee correct kernel operation, as shown by return-to-libc [30] and return-oriented programming [17, 28] attacks that make use of unmodified, non-malicious code to accomplish malicious goals. Kernel code integrity is the foundation on which the more stringent guarantees of control-flow integrity [1] and data integrity [4, 8, 11] are layered.

Unlike OSck, that maintains the guest OS integrity against malicious applications, Overshadow [9] ensures secrecy and integrity for application memory against a compromised OS kernel. Overshadow enforces that application code and data pages appear encrypted to the OS, thus ensuring application integrity. However, Overshadow does not protect guest OSes from corruption, hence it is possible for applications to receive incorrect yet consistent responses from a compromised OS. For example, an application may believe it is reading a configuration file and yet actually be reading a fabricated version.

7.1 Control-flow integrity

Petroni and Hicks [23] describe a VMM-based approach to maintaining control-flow integrity (execution follows a predetermined control-flow graph [1]) They perform periodic scans of kernel memory, verifying *state-based control-flow integrity*, a property similar to that verified by OSck. They compute a type graph from kernel source, and use this graph to determine a set of global roots for kernel data, the fields of kernel data structures that are function pointers, and the fields that point to other structures. Periodically, their integrity checking monitor executes and traverses the kernel's object graph starting from the set of global roots, successively following pointers to examine all allocated kernel data structures. For each object, the system verifies that all function pointers have desired properties, for example, that they point to a specific set of allowed functions. This requires a breadth-first search of the data structure graph, while OSck verifies type safety via a linear scan of kernel memory.

The Petroni/Hicks system does not consider the stack, though they assert that attacks on the stack are generally transient. Our "return to schedule" rootkit demonstrates a non-transient stack-based attack that would not be caught by their system. Furthermore, their system does not address inconsistencies that can occur because their checks are performed as they scan through a tree of kernel data structures that can be concurrently modified by the OS. OSck avoids these inconsistencies by quiescing when integrity is in question after preliminary checks.

HookSafe [32] protects function pointers (often called hooks) by moving them to a write-protected area. Instructions that access these pointers within the kernel are detected with a profiling run, and then rewritten via binary translation to access the new hook locations. Any accesses to function pointers not present in the profiling run are not correctly translated during runtime. OSck verifies a larger class of integrity properties than properties about kernel function pointers. When possible, it protects function pointers by placing them in hardware-protected memory (whose access policy is determined by the hypervisor), otherwise, it periodically verifies that function pointers point to legitimate code targets according to a flexible policy.

7.2 Data Integrity

KernelGuard [25] protects kernel data structures by selectively interposing on memory writes. KernelGuard checks the kernel function that writes a guarded memory region against explicit sets of allowed functions per region. While specifying what functions can modify which regions is a powerful primitive, it does not prevent large classes of attacks where (for example) a function that is permitted to modify a region does so maliciously. For example, protocol handler and netfilter filter registration functions need to be able to install new filters—the latter is used in the firewalling tool `iptables`. OSck enforces OS integrity guarantees on the data structures themselves—in this case, that the functions for protocol handlers and netfilter filters are contained within kernel or allowed module text. Indirect guarantees do not ensure OS integrity.

LiveDM [18] is a system for tracking the types of dynamically allocated kernel memory. It operates by interposing on guest VM operation, waiting until the VM reaches any of a set of kernel memory allocation functions. For these functions, the VMM keeps a "shadow stack" of pending return addresses that it uses to determine when a particular call returns so it can associate call sites (kernel code locations) with return values. The types of return values can then be inferred by static analysis of surrounding code regions; an instrumented version of `gcc` is used to generate an AST for static analysis. While OSck focuses on typing slab allocation, it only uses CIL for static analysis and does not require instrumenting a compiler. No performance measurements are given for LiveDM, but OSck's cache scanning mechanism has low system impact.

Loscocco et. al. present the Linux kernel integrity monitor (LKIM) [21] which is a hypervisor component that intelligently hashes kernel data structures for integrity. LKIM does not provide advanced features like type-checking the kernel heap or providing an interface to kernel programmers to write their own integrity checks.

8. Conclusion

OSck is a virtual machine based system that detects rootkits by determining violations to operating system invariants. OSck introduces new mechanisms for bulk verification of types, and consistent data structure views to provide practical and efficient detection. Our deployment of OSck can efficiently detect many kinds of rootkits including two new classes that cannot be handled by any previous rootkit detection system.

Acknowledgments

We thank our shepherd, Mendel Rosenblum, as well as the anonymous reviewers for their comments on earlier versions of this paper. This research is supported by NSF CNS-0905602, NSF CAREER CNS-0644205, the NSF Graduate Research Fellowship Program, and a Google research award.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov 1996.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65, Washington, DC, USA, 1997. IEEE Computer Society.

[4] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 246–251, Washington, DC, USA, 2007. IEEE Computer Society.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[11] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Procedings of the Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[14] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[15] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, February 1988.

[17] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

[18] D. X. Junghwan Rhee. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical report, 2 2010.

[19] B. Kauer. OSLO: improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 16:1–16:9, Berkeley, CA, USA, 2007. USENIX Association.

[20] A. Kivity. kvm: The Linux Virtual Machine Monitor. In *The 2007 Ottawa Linux Symposium*, OLS '07, pages 225–230, July 2007.

[21] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 21–29, New York, NY, USA, 2007. ACM.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[23] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[24] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM.

[25] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. *International Conference on Availability, Reliability and Security*, 0:74–81, 2009.

[26] J. Rutkowska. Linux kernel backdoors and their detection. In *ITUnderground*, Oct 2004.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[28] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[29] A. Shevchenko. Rootkit evolution. `http://www.securelist.com/en/analysis?pubid=204792016`.

[30] Solar Designer. Getting around non-executable stack (and fix). 1997. http://seclists.org/bugtraq/1997/Aug/63.

[31] Trusted Computing Group. *TPM Main Specification*, 2007. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[32] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[33] D. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2001.