Operating System Interface Obfuscation and the Revealing of Hidden Operations

Abhinav Srivastava[1] Andrea Lanzi[2] Jonathon Giffin[1] Davide Balzarotti[2]

[1]School of Computer Science, Georgia Institute of Technology
[2]Institute Eurecom
{abhinav,giffin}@cc.gatech.edu, {andrew,davide}@iseclab.org

Abstract. Many software security solutions—including malware analyzers, information flow tracking systems, auditing utilities, and host-based intrusion detectors—rely on knowledge of standard system call interfaces to reason about process execution behavior. In this work, we show how a rootkit can obfuscate a commodity kernel's system call interfaces to degrade the effectiveness of these tools. Our attack, called *Illusion*, allows user-level malware to invoke privileged kernel operations without requiring the malware to call the actual system calls corresponding to the operations. The Illusion interface hides system operations from user-, kernel-, and hypervisor-level monitors mediating the conventional system-call interface. Illusion alters neither static kernel code nor read-only dispatch tables, remaining elusive from tools protecting kernel memory. We then consider the problem of Illusion attacks and augment system call data with kernel-level execution information to expose the hidden kernel operations. We present a Xen-based monitoring system, *Sherlock*, that adds kernel execution *watchpoints* to the stream of system calls. Sherlock automatically adapts its sensitivity based on security requirements to remain performant on desktop systems: in normal execution, it adds 1% to 10% overhead to a variety of workloads.

1 Introduction

Honeypots and other utilities designed to audit, understand, classify, and detect malware and software attacks often monitor process' behavior at the system-call interface as part of their approach. Past research has developed a widespread collection of system-call based security tools operating at user or kernel level [7, 10, 14, 22, 30, 33] and at hypervisor level [6, 11, 23]. Employing reference monitors at the system-call interface makes intuitive sense: absent flaws in the operating system (OS) kernel, it is a non-bypassable interface, so malicious code intending to unsafely alter the system will reveal its behavior through the series of system calls that it invokes.

Current malware often makes use of kernel modules, also known as rootkits, to conceal the side-effects of a malicious user-level process. Rootkits are attractive for attackers because they combine a high-privilege kernel component with an easy-to-use, and easy-to-update application. This trend is supported

by a recent report from the Microsoft Malware Protection Center [37], that estimates that Rootkits are responsible for around 7% of all infections reported from client machines. Many rootkits, for example *adore* [24] and *knark* [25], provide functionalities to hide processes, network connections, and malicious files by illegitimately redirecting interrupt or system call handling into their kernel modules. However, this class of attacks can be prevented by making the interrupt descriptor table (IDT), system service descriptor table (SSDT), system-call handler routines, and kernel dynamic hooks write protected, as proposed by Jiang et al. [11, 44]. Systems like above assume that protections against illegitimate alteration of these objects will force malicious software to follow the standard system-call interface when requesting service from the kernel.

Unfortunately, this assumption does not hold true. In the first of two principal contributions of this paper, we show how a rootkit can obfuscate a commodity kernel's system-call interface using only *legitimate functionality* commonly used by benign kernel modules and drivers. In particular, we present an attack, *Illusion*, that allows malicious processes to invoke privileged kernel operations without requiring the malware to call the actual system calls corresponding to those system operations. In contrast to prior attacks of the sort considered by Jiang et al. [11, 44], Illusion alters neither static kernel code nor kernel dispatch tables such as the IDT or SSDT. As a consequence, during the execution of malware augmented with the Illusion attack, an existing system-call monitor would record a series of system operations different from the ones actually executed by the malware.

Using rootkits augmented with the Illusion's obfuscation capability, attackers can execute malicious behaviors in a way that is invisible to most of the current security tools and malware analyzers that monitor the processes behavior at the system call level [6, 18, 21]. And since Illusion does not make any changes to kernel code and data structures, it cannot even be detected by Kernel integrity checkers [28].

The Illusion attack is possible because current analyzers depend on the standard system-call interface to represent the underlying changes that a process is making to the system. Importantly, they do not take into account the actual execution of the requested system call inside the kernel. In our second principal contribution, we present a defensive technique to detect Illusion attacks and recover the original system calls that the malware would have invoked had the Illusion attack been absent. To demonstrate the feasibility of our defense, we have developed a prototype system called *Sherlock* using the Xen hypervisor and a fully-virtualized Linux guest operating system. Sherlock tracks the kernel's execution behavior with the help of *watchpoints* inserted along kernel code paths that are executed during the service of a system call. Mismatches between the system call invoked and the service routine's execution indicate that an Illusion attack has altered the internal semantics of the system-call interface.

Sherlock is an adaptive defensive system that tunes its own behavior to optimize performance. It is important to note that Sherlock itself does not provide detection capability for attacks other than the Illusion attack; rather, it augments

existing system call analyzers with information about hidden kernel operations. When watchpoints match the expected execution of the requested system call, then traditional system call monitoring indeed observes the correct events. When Sherlock detects the Illusion attack, however, a system call monitor records a faulty series of system calls. Sherlock will then assist the monitor by switching into a deep inspection mode that uses the execution behavior of the kernel to reconstruct the actual system call operations executed by the malware. As soon as the malicious invocation of the system call completes, Sherlock switches back into its high performance mode.

During benign execution with watchpoints enabled, Sherlock imposes overheads of 10% on disk-bound applications, 1%–3% on network-bound software, and less than 1% on CPU-bound applications.

In summary, this paper makes the following contributions:

- It presents the Illusion attack, a demonstration that malicious software can obfuscate a commodity kernel's system-call interface using only the legitimate functionality used by benign modules and drivers. The malicious software controls the obfuscation, so every instance of malware using the Illusion attack can create a different system call interface.
- It discusses the design and implementation of Sherlock, a hypervisor-based kernel execution monitor. Sherlock uses watchpoints and models of system call handler execution behavior to reconstruct privileged operations hidden by an Illusion attack.
- It shows how our adaptive kernel execution monitoring technique balances security and performance by deeply inspecting the kernel state only when the kernel executes hidden operations.

2 Related Work

If defenders know *a priori* about offensive technologies used by attackers, then they can develop appropriate remedies. To this end, researchers have performed various attack studies to better understand how attackers can evade host-based security tools. Mimicry attacks [41, 43] against application-level intrusion detection systems [10] escape detection by making malicious activity appear normal. Baliga et al. [2] proposed a new class of kernel-level stealth attacks that cannot be detected by current monitoring approaches. David et al. [5] created a rootkit for the ARM platform. The rootkit was non-persistent and only relied on hardware state modifications for concealment and operation. We apply this style of attack reasoning to system-call monitors. Like previous literature, we also assume the perspective of an attacker who is trying to undetectably execute malicious software. By taking this view, we hope to help defenders understand and defend against the threat of system call API obfuscation. Our obfuscation does not require any modification to existing operating systems' code and data, in contrast to earlier attacks that modified the system-call or interrupt tables [4, 17].

We use a hypervisor to observe the behavior of a kernel executing within a virtual machine. Jones et al. [12] developed a mechanism for virtualized environments that tracks kernel operations related to specific process actions. With their system, process creation, context-switch, and destruction can be observed from a hypervisor. Srivastava et al. [39] created a system that monitors the execution behaviors of drivers. In contrast, Sherlock allows for arbitrary kernel behavior tracking based upon watchpoints inserted into relevant kernel code execution paths.

Insertion of additional state exposure events into existing code has been a recurring idea in past research [20, 40]. Payne et al. [26] developed a framework that inserted arbitrary hooks into an untrusted machine and then protected those hooks using a hypervisor. In a similar way, Windows filter drivers allow hooking and interception of low-level kernel execution events. Giffin et al. [9] used null system calls to improve the efficiency of a user-level system-call monitor. Xu et al. [45] inserted waypoints into applications; execution of a waypoint adds or subtracts privilege from an application so that it better adheres to access restrictions based on least privilege. Sherlock uses its watchpoints to gain a view of the system call handler execution behavior, and it expects that these watchpoints will be protected via standard non-writable permissions on memory pages containing kernel code.

Sophisticated attackers may attempt to bypass the portions of kernel code containing watchpoints by altering execution flow in a manner similar to that used to execute application-level mimicry attacks [15]. These attacks abuse indirect control flows and could be prevented using techniques such as control flow integrity [1, 29, 35] and kernel hooks protection [44] that guard indirect calls and jumps. Sherlock does not provide such protections itself.

3 System Call Obfuscation via Illusion Attacks

3.1 Motivation

In the first part of the paper we adopt an offensive role and target system-call based monitoring software such as those used to analyze malware, track information flows among processes, audit suspicious execution, and detect attacks against software. The goal of this work is to investigate attackers capabilities to hide the behavior of their malicious code executing on the users' systems.

In this section we present the Illusion attack, a powerful way to extend the functionality provided by rootkits to enable them to blind system-call based detectors. An alternative way of defeating these detectors would be to design malware completely running in kernel-space. In this case, a malicious driver would perform all activities using the kernel code, hence bypassing the existing system-call interface. Though this approach may seem attractive, as shown by Kasslin [13], attackers still prefer a rootkit-based design because it is easier to develop, use, and maintain [32].

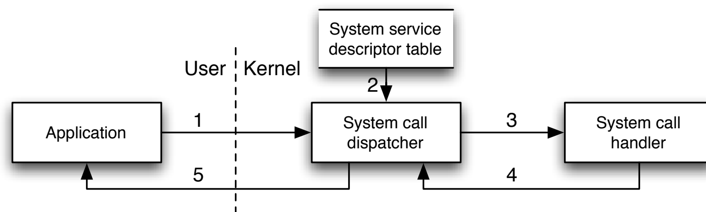

Fig. 1. Normal System-call Execution Path

3.2 Abilities of the Attacker

We use a threat model that reflects current, widespread attacks and state-of-the-art monitoring systems. To launch the Illusion attack, the attacker must install a malicious kernel module or driver; this may occur by acquiring a signature for the module, loading the module on a system that does not check signatures, loading the code via an exploit that circumvents module legitimacy checks [3], or enticement of a naïve user. A legitimate signature is the most capable technique, as it evades cutting-edge defenses against kernel-level malicious code [31]. This fact is demonstrated by the recent Stuxnet worm that stole the digital signature of two companies to install its malicious driver [16, 27]. The loaded module can access and alter mutable data in the kernel, but we assume that it cannot modify static kernel code and data because those changes can be detected by current monitoring systems [34]. This threat model is sufficient to allow the design of our Illusion attack, though we will revisit this model with additional expectations when presenting the Sherlock defensive system.

3.3 Attack Overview

Illusion attacks obfuscate the sequence of system calls generated by a malicious process requesting service from the kernel. The malicious process must still receive the desired service, else the malware would fail to have an effect beyond CPU resource consumption. The Illusion attack must enable the malicious process to execute the same system call requests that it would make in the absence of the Illusion attack, but the particular calls should not be revealed to a system-call monitor.

Figure 1 depicts normal in-kernel system call dispatch with the following steps. (1) When an application invokes a system call, it issues a software interrupt or system call request. The CPU switches from user to kernel mode and begins executing the system call dispatch function. (2) The dispatch function reads the system call number, which indicates the type of system service requested, and uses it as an index into a table of function pointers called the system service descriptor table (SSDT). Each entry in the SSDT points at the system call handler function for a particular system call. (3) After reading the pointer value

from the SSDT, the dispatch function transfers execution to that target system call handler in the kernel. (4) When the handler completes, it returns its results to the dispatch function, which then (5) copies them back into the application's address space and returns control to the application. We call this flow the *normal system-call execution path*. An attack that alters system call dispatch—steps 2 and 3—will not succeed, as our threat model does not allow an attacker to alter the entries in the SSDT without detection.

However, a number of system calls allow *legitimate* dispatch into code contained in a kernel module or driver. Consider `ioctl`: this system call takes an arbitrary, uninterpreted memory buffer as an argument and passes that argument to a function in a kernel module that has registered itself as the handler for a special file. Benign kernel modules regularly register handler functions for such files; a malicious module performing the same registration exhibits no behaviors different from the benign code. However, a call to `ioctl` will be directed into the malicious module's code together with the buffer passed to `ioctl` as an argument.

In our construction, this argument contains the original system call request in a serialized form. The malware will marshal the original system call into a memory buffer that it subsequently passes to the `ioctl` system call. The handling function within a malicious kernel module will unmarshal the buffer to identify the actual operation requested by the malware and will then directly call the system call handler for that operation. With this interface illusion in place, the kernel still executes the same operations that the malware instance would have executed without the obfuscation. However, system call monitoring utilities would observe a sequence of `ioctl` requests and would not realize that malicious operations had occurred.

This attack has properties appealing to an attacker:

- The attacker can select from a collection of mechanisms offering legitimate dispatch to module code, such as shared memory, netlink sockets, the proc filesystem, virtual filesystem (vfs) dispatch, or a character device. A particular instance of an Illusion attack may use any or all of these operations in combination as replacements for existing system calls.
- The attacker controls marshaling and unmarshaling operations. Although a system call monitor knowledgeable of a system call marshaling format could recover the original system call sequence by unmarshaling arguments, the attacker can continually change the marshaling format to prevent the monitor from gaining this knowledge. This continual change is similar to the effect of polymorphic engines or packers upon malicious code.
- The attacker does not need to obscure all system calls; she only needs to obscure the set of system calls performing operations that would reveal the malicious intent of her malware to security tools that reason about the malicious behavior. Other system calls executed by the malware, including dummy calls that could be inserted for confusion or mimicry, need not be obfuscated. This reduces the rate of calls to the Illusion attack's dispatching

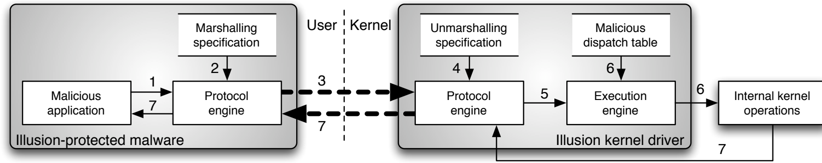

Fig. 2. Alternative System-call Execution Path

operation and increases the apparent normalcy of the system call sequence, making the attack stealthier.
– The attacker can automatically convert a malware instance invoking traditional system calls into a malware instance using an Illusion attack. A straightforward source-to-source transformation can simply replace a malware's system calls with a marshaling operation and a call to an Illusion dispatching system call. Similar automated transformation could occur at the binary level at any system call site whose arguments can be recovered via static binary analysis.
– The attacker could define new system calls executable through the Illusion attack but having no counterpart in the traditional system-call interface.

Although a system-call monitor is able to know that system-calls are occurring, it cannot evaluate the semantic meaning of the operations.

3.4 The Illusion Kernel Module

The Illusion attack creates an *alternative system call execution path* that executes system operations requested by malicious applications. By using an alternative execution path, malware is able to perform any system operation without raising the actual system call event that normally invokes that operation. Our design uses a kernel module, a cooperating malware instance, and a communications channel between the two. The kernel module is composed of two engines: the protocol engine and the execution engine. The protocol engine unmarshals an argument buffer into a system operation request and interprets the semantics of that request. The execution engine actually executes the requested system call operation. The execution engine can execute the system operations in several different ways, including: (1) a direct call to the corresponding system-call handler, (2) a direct call to any exported kernel function with similar behavior, or (3) execution of code within the kernel module followed by a jump into low-level portions of the kernel's handler, such as device driver code that accesses hardware of the computer.

Figure 2 shows how the alternative system-call execution path alters a kernel's processing of system call operations. (1) Malware sends an operation request to the protocol engine. (2) The protocol engine obfuscates the request by using the marshaling specification. (3) The malware sends the obfuscated request to

the kernel module via the particular communication channel used by this instance of the Illusion attack. (4) The protocol engine in the kernel receives the request, unmarshals the argument using the specification, and (5) sends the requested operation to the execution engine. (6) The execution engine invokes the appropriate kernel operation, using one of the three execution methods described above. (7) When the function returns, it passes the results to the protocol engine which then returns them to the malware. If needed by the malware, the protocol engine can obfuscate the response, which would then be deobfuscated by the corresponding engine in the user-level malware.

3.5 Implementations

We have built two different prototypes of the Illusion attack: one for Windows and one for Linux. In order to validate the effectiveness of the attack to obfuscate system call sequences, we also developed *Blinders*, a Qemu-based system-call tracer for Windows designed using the same principles of Jiang and Wang [11]. In the rest of the section we describe the details of the Windows implementation. However, the Linux code follows a nearly identical design.

Our Windows Illusion attack uses the `DeviceIoControl` system call as its entry point into the kernel. This function exchanges data between a kernel driver and an application through a device. It receives parameters that include a device handle for the device and a memory buffer to pass information to the driver. We used this buffer to pass serialized system operation requests according to the marshaling protocol.

The execution engine within our Windows kernel driver implements all three execution methods described in Section 3.4. For method 1, the kernel driver prepares the stack layout according to the requirements of the system-call handler function. After extracting the handler's function pointer from the system-call dispatch table, it directly calls the handler. For the second method, the driver directly calls exported kernel functions; for example, a call to the function `ZwCreateFile` performs a file creation operation. The last method is more subtle in that the driver executes some part of the code inside itself and only then jumps into the kernel code midway along the control-flow path of the requested operations. We copied a piece of the original function's assembly code from the kernel into the driver as inlined assembly. We equipped our kernel driver to execute two operations: creation of a file and creation of a registry key.

For our experimental marshaling protocol, the user-space malware constructed a sequence of comma-separated fields. We defined two static fields: the first contained the unique number for a system operation, and the second contained a value that identified the number of parameters for the operation. All subsequent values represented the values of parameters.

We executed a prototype malware instance using our Illusion attack together with Blinders inside Windows XP. We issued a request from the user-space malware component for file and registry key creation. The system calls issued by the user-space component were identified by Blinders as the incorrect call,

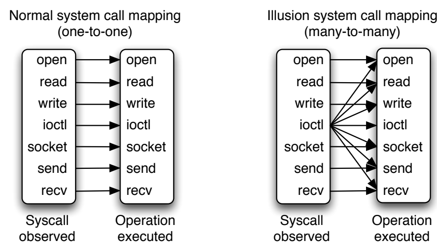

Fig. 3. System calls and examples of system call-to-operation mappings.

as expected. The system-call tracer was only able to see the system call used to communicate with the driver, which was `DeviceIoControl` in this experiment.

We implemented a similar attack prototype for the Linux 2.6 kernel. In this version, the kernel module directly called exported kernel functions to perform requested operations, and it provided unmarshaling for open, read, write, socket, connect, send, and receive requests. The attack used `ioctl` as the hidden system call entry point to the Illusion dispatch functionality. We ran the Linux prototype, executed all operations, and used the standard system call logging tool *strace* to observe the issued system calls. When an experimental malware instance executed any of the hidden calls, *strace* only showed the `ioctl` system call in its log.

3.6 Foundation

The Illusion attack can be understood by reasoning about system call mapping operations. In a correct trace, there is a one-to-one mapping between a system operation invoked by a process and the operation observed by the monitor. The Illusion attack breaks trace correctness by creating a new many-to-many relation, unknown to a monitor, between the system call invoked and the real system operations executed.

Figure 3 shows two different mappings between a system operation observed by interposing on the normal system call mechanism and the real system operation executed within the kernel. The first mapping is a normal one-to-one relation, where for each system-call invocation there is only one corresponding system operation executed by the kernel. The second mapping, many-to-many, is an example mapping created by the Illusion attack and not known to the monitor. In fact, even if the monitor were aware that system calls were obfuscated through such a relation, it would be unable to reverse the many-to-many relation without additional information to disambiguate among the collection of actual system calls that all map to the same observed call.

3.7 Discussion

Our attack diminishes the ability of system call monitors to understand actual malicious process behavior.

Hypervisor-based system-call analyzers: Due to the prevalence of kernel malware, hypervisor-based system-call monitoring tools have been proposed [11,23]. Although these tools provide tamper-resistance, they still depend on the normal system-call execution path to record system operations. Illusion is able to obfuscate the system call mechanism and will hide the system operations from these tools.

Malware unpackers and analyzers: A similar effect can be seen on system-call based malware unpacking and analysis [6,18]. Martignoni et al. [18] proposed an unpacker that monitored the execution of suspected malware and assumed that the code was unpacked when the application issued a dangerous system call. In another work, Martignoni et al. [19] presented a behavioral monitoring system that relied on system calls. The system intercepted system calls as a low-level event and tried building high-level behavioral model. Since the Illusion attack hides the malicious calls, the system would never be able to build high-level model.

Anti-virus tools and intrusion detectors: Some anti-virus software and IDS tools [21] use the system-call interception mechanism to detect a malware infection. Such tools modify the system-call dispatch table in order to intercept the system calls. Even in this case, the Illusion attack is able to hide the system call information without triggering the sensors set by the security application.

Many common security tools such as kernel memory scanners or system-call based anomaly detectors do not detect Illusion attacks. The Illusion attack does not modify a kernel's static data structures or dispatch tables, which allows it to remain stealthy to security tools that monitor these data. Illusion augments malicious software that would not be protected by application-specific anomaly detection systems. Importantly, an attacker can always bolster an unusual system call sequence of Illusion requests with unobfuscated nop system calls to create an undetectable mimicry attack sequence.

4 Sherlock

We developed a hypervisor-based prototype system called *Sherlock* that detects the presence of an Illusion attack and determines the actual system operations executed via the hidden interface. It is designed to achieve the following goals:

- **Secure system:** Sherlock provides security by exposing hidden operations happening inside the operating system. It uses watchpoints, or state exposure operations within kernel code, to track kernel execution. The watchpoints reveal the hidden behaviors of an Illusion attack's kernel module executing within the kernel.
- **Tamper resistant:** Attackers controlling the operating system should not be able to alter Sherlock's execution. It uses a hypervisor-based design to remain isolated from an infected system. It may protect its watchpoints inside the guest OS by making the kernel's static code write-protected to prevent tampering [34].

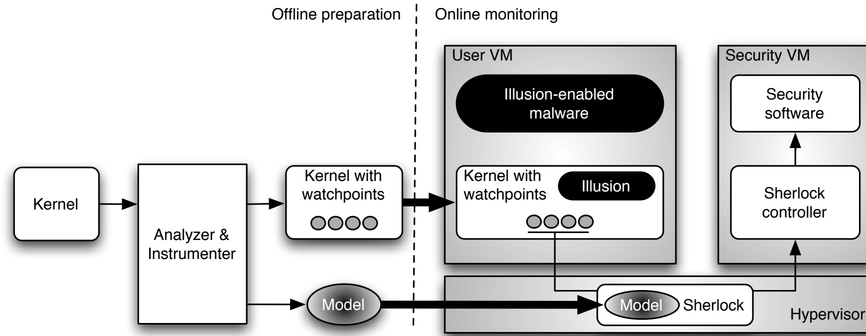

Fig. 4. Sherlock's Architecture.

– **High performance:** Sherlock should not unreasonably slow the execution of benign software. Its design permits adaptive execution, where performance-costly operations will be invoked only when an Illusion attack may be underway.

Our system-call API obfuscation detection operates in two phases. In the first phase (offline preparation), we analyze the kernel source code and insert watchpoints along kernel code paths that execute during the service of a system call. We create models of system-call handler execution behavior to detect Illusion attacks. The second phase (online monitoring) divides the system into three components: an untrusted user virtual machine (VM), a trusted security VM, and a modified Xen hypervisor (Figure 4). The user VM runs a guest operating system (OS) instrumented with watchpoints; our prototype implementation uses Linux. Each watchpoint notifies the hypervisor-level Sherlock monitor of kernel execution behavior via a `VMCALL` instruction, and Sherlock determines if the watchpoint is expected or suspicious. Suspicious watchpoints are passed to the security VM for deep analysis.

4.1 Threat Model

We expand on the threat model previously described in Section 3.2. As Sherlock uses virtualization, we assume that an attacker is unable to directly attack the hypervisor or the security VM, an expectation that underlies much research in virtual machine based security. Sherlock relies on kernel code instrumentation, and so we expect that an Illusion attack's kernel module will be required to call into the existing instrumented code of the kernel at some point in its handling of hidden operations. Our belief is that while a module may carry some fragments of duplicated kernel code containing no watchpoint instrumentation, it cannot predict all possible configurations of hardware and file systems and must eventually make use of the existing code of a victim's instrumented kernel. Although

Operation	*Functions*
Open	sys_open, sys_open, open_namei
Read	sys_read, sys_read, do_sync_read generic_file_aio_read
Write	sys_write, sys_write, do_sync_write, generic_file_aio_write
Socket	sys_socket, sys_socket, _sock_create, inet_create, tcp_v4_init_sock
Connect	sys_connect, sys_connect, net_stream_connect, tcp_connect_init
Send	sys_send, sys_send, __sock_sendmsg, tcp_sendmsg
Receive	sys_recv, sys_recv, __sock_recvmsg, tcp_recvmsg

Table 1. Watchpoint Placement (Linux 2.6)

malicious modules could read and duplicate the kernel's existing code, W⊕X protections would prevent them from executing the copied code. While these protections do not address the problem of object hiding via direct kernel object manipulation (DKOM), such attacks are not Illusion attacks and would be best detected with kernel integrity checkers [46].

4.2 Exposing Kernel Execution Behavior

Sherlock monitors the kernel's execution behavior with the help of watchpoints inserted along the kernel code paths. These watchpoints expose kernel-level activities and are implemented using the `VMCALL` instruction. In order to distinguish among watchpoints, each watchpoint passes a unique identifier to Sherlock's hypervisor component that helps Sherlock identify the operation being performed. For example, in the case of a *read* operation in Linux, a watchpoint on the execution path of the read system-call handler sends a unique watchpoint `do_sync_read` whenever it executes. When the hypervisor component receives the `do_sync_read` watchpoint, it infers that a read operation is being performed.

We chose where to place watchpoints by performing a reachability analysis of kernel code. Our analysis finds all functions reachable from the start of the system-call handler down to the low-level device drivers. This analysis yields a set of functions that may legitimately execute subsequent to the entry of the system call handler. We anticipate that inserting watchpoints in all these functions would adversely impact the performance of the operating system. We instead find a set of functions that dominate all other functions in the reachability graph, with separate dominator functions chosen from separate OS components. A function v in the call graph for a system-call handler dominates another function w if every path from the beginning of the call graph to function w contains v. We define a component as a (possibly) separately-linked code region, namely the core kernel and each distinct module. For example, the components involved in the execution of a *read* operation are the core system-call handler, the filesystem driver, and the disk driver. We insert a watchpoint in the middle of each selected dominator. Continuing the *read* example, we insert watchpoints in `sys_read`, `do_sync_read` and `generic_file_aio_read`. Table 1 shows the list of operations and corresponding functions where watchpoints were placed.

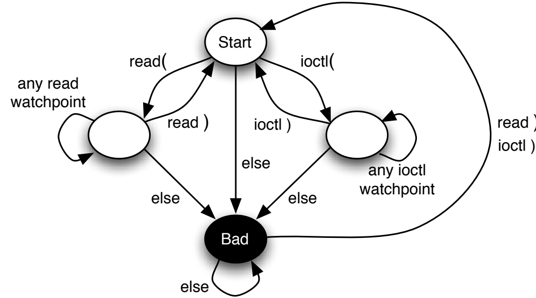

Fig. 5. A portion of Sherlock's Büchi automaton used to distinguish between benign and suspicious operations. All states are accepting. The figure shows the portions of the automaton corresponding to the system calls `read` and `ioctl`; the patterns for other system calls are similar. Whenever the automaton enters the *Bad* state, the internal kernel operations are not matched with a requesting system call, an indicator that an Illusion attack is underway.

We perform this analysis by inspecting the source code of the kernel. During this process, we insert watchpoints inside the kernel functions corresponding to different system calls and ran user-space applications to stimulate our watchpoints. This process has given us confidence that the watchpoints are at the correct locations. To reduce errors that may be introduced by the manual analysis, an automated approach is desirable. Previous research has already performed automatic analysis of the kernel source code to find locations of hooks and security specifications [8, 42], and these systems can also be suitable for selection of Sherlock watchpoint instrumentation locations. For these reasons, we do not consider such analysis as our contribution, but we built a detection system on the top of the output of such analysis.

Sherlock depends upon watchpoints to know about the state of the executing kernel. To defeat attacks in which an attacker may try to confuse Sherlock by executing fake `VMCALL` instructions, Sherlock identifies the code position (instruction pointer) from where a `VMCALL` instruction is invoked. With this knowledge, Sherlock discards `VMCALL` instruction that does not correspond to the set of our watchpoints.

4.3 Modeling System Call Handler Execution

Sherlock's goal is to reveal hidden system operations. It must distinguish between a normal system call and a suspicious system call. A call is suspicious if it was altered by a many-to-many mapping, as described in Section 3.6. We detect altered system call operations by comparing the sequence of watchpoints executed by the running kernel against an automaton model describing the expected behavior of each system call handler function in the kernel.

To know when system calls are legitimately invoked by an application, we insert two watchpoints in each system-call handler function to know the start and end of the system operation. When a benign application invokes a system

operation, execution reaches the system-call handler along the normal execution path and the first watchpoint in that handler executes. Sherlock's hypervisor component receives the watchpoint and traverses an opening bracket "(" transition indicating which system call was invoked. It then receives subsequent watchpoints that had been inserted along the control-flow path of the system operation. Every time it receives a watchpoint, it verifies that the watchpoint was expected given the invoked system call. It performs this step repeatedly until it sees the watchpoint ")" corresponding to the end of the system call request. However, if any received watchpoint should not have occurred given the system call request, then an Illusion attack may have obfuscated the system call interface.

The language of allowed watchpoint sequences is omega-regular, so a finite-state automaton can easily characterize the expected and suspicious sequences. Figure 5 shows a portion of the automaton used by Sherlock to distinguish between benign and suspicious operations, with all individual system calls aggregated into a single model. Whenever a watchpoint executes and is not part of the system operation, Sherlock will take the *else* transition and reach the *bad* state. At the bad state, any operation can be executed until the system call completes, taking Sherlock back to the *start* state. A kernel runs forever, so Sherlock's model is a Büchi automaton with all states final.

4.4 Adaptive Design

Sherlock tunes its performance to match the current security condition of the monitored kernel. Its hypervisor-level component receives watchpoints from the guest operating system and operates the Büchi automaton to identify the suspicious activities. If the automaton is not in a suspicious state, then the kernel is executing in an expected way. Sherlock's hypervisor component immediately discards all received watchpoints, as they would provide no additional information to a traditional system call monitor at the standard interface given the absence of an Illusion attack. Otherwise, Sherlock concludes that this is a suspicious activity and starts logging subsequent watchpoints until the conclusion of the system call. During this process, it pauses the user VM and extracts parameters involved in the operation at each watchpoint. This data gathering helps Sherlock rebuild information about the actual system operation underway. Sherlock unpauses the user VM when an analyzer in the security VM completes the inspection of the state of the VM.

The security VM has a user-level controller process that controls Sherlock's operation and runs infrequent, performance-costly operations. The controller starts and stops the user VM as needed to perform system call identification during an active Illusion attack. The controller passes attack information to existing security software, pauses the user VM, performs detailed analysis of the unexpected watchpoint, and then restarts the guest.

Consider again our example of the *read* system call and corresponding in-kernel execution behavior. Suppose malware uses the Illusion attack to invoke the *read* via arguments marshalled through the *ioctl* system call. The Illusion

kernel module may invoke the read operation by directly calling the filesystem driver's read function `do_sync_read`. As Table 1 indicates, we inserted a watchpoint in this function. Sherlock receives the beginning marker event "`ioctl(`" followed by the watchpoint `do_sync_read`. Since these events correspond to two different operations, they do not match and the automaton enters the *Bad* state. Sherlock's hypervisor component reports this mismatch to the controller along with an indication that *read* is believed to be the actual system call operation underway. This report activates the controller's deep analysis mode – pause the user VM, extract the information from the user VM, and pass the computed system call to higher-level monitoring utilities and security software in the security VM.

5 Evaluation

We paired Sherlock with the Illusion attack to measure its impact on a typical system. We first provide an empirical false positive analysis of Sherlock in Section 5.1. We expect Sherlock to slightly diminish system performance and carry out experiments measuring its effect in Section 5.2. Finally, we discuss future work to address the current limitations of Sherlock in Section 5.3.

5.1 False Positive Analysis

We empirically analyze the ways in which Sherlock may erroneously believe interface obfuscation is underway or fail to recognize the presence of an Illusion attack. A false positive may arise if our model of kernel system call handlers is incorrect and omits allowed behavior. For example, Linux uses the `write` system call for both writing to a file and writing to a socket. If a `write` is executed to write to the socket, it invokes watchpoints both corresponding to the write operation and socket operation. In this case, Sherlock would observe watchpoints corresponding to two different system operations. Even though a naïve automaton model would cause Sherlock to conclude that this is suspicious activity, simple refinements to the model can remove these false positives and improve Sherlock for all future use of the system.

5.2 Performance

As a software mechanism that intercepts events that may occur at high rates, we expect Sherlock's monitoring to impact the execution performance of guest applications. To measure its performance, we carried out several experiments involving CPU-bound, disk-bound, and network-bound workloads. For all experiments, we used Fedora in both the security VM and user VM running above Xen 3.0.4 operating in fully virtualized (HVM) mode. Our test hardware contained an Intel Core 2 Duo processor at 2.2 GHz, with VT-x, and with 2 GB of memory. We assigned 512 MB of memory to the untrusted user VM. All reported

Operation	*Normal VM Time (μs)*	*Sherlock Time (μs)*	*Sherlock + Illusion Time (μs)*	*# of Watchpoints Inserted*	*Executed*
Open	17.946	26.439	61.752	3	1
Read	19.419	30.537	60.970	4	1
Write	27.025	37.807	90.656	4	2
Socket	24.515	45.558	115.106	5	3
Connect	1879.894	1905.336	1984.419	4	2
Send	717.391	746.416	838.440	4	2
Receive	8.377	20.958	79.488	4	2

Table 2. Single operation processing costs

results show the median time taken from five measurements. We measured microbenchmarks with the x86 `rdtsc` instruction and longer executions with the Linux `time` command-line utility.

We measured the time to process a single watchpoint both with and without an Illusion attack underway. Whenever a watchpoint executes, the control reaches the hypervisor and Sherlock's hypervisor component checks whether or not the watchpoint is due to a hidden operation. If so, it then pauses the guest VM and sends information to the Sherlock controller. Otherwise, it discards the watchpoint and sends control back to the VM. With no Illusion attack underway, each watchpoint cost 3.201μs of computation. With the attack, Sherlock's per-watchpoint cost increased to 39.586μs. The adaptive behavior of Sherlock is well visible: its watchpoint processing is fast for the benign case and an order of magnitude more costly when deep inspection is performed due to the detection of a hidden operation. We note that hidden operation processing shows a best-case cost, as our experiment performed a no-operation inspection into the guest OS. An analyzer performing a more detailed analysis may take more time before resuming the guest VM's execution.

Sherlock expects multiple watchpoints for each system call. We next measured the time to execute a single system call in the presence of watchpoints. We measured the time to execute `open`, `read`, `write`, `socket`, `connect`, `send`, and `recv`, as these correspond to the handlers that we instrumented. We wrote a sample test program to invoke each of these calls and measured execution times in three ways: without Sherlock, with Sherlock during benign execution, and with Sherlock during an Illusion attack. Table 2 presents the result of this experiment, and it also shows the number of watchpoints inserted for each operation and the number of watchpoints executed during an Illusion attack. It can be seen from the table that for `open` and `read` operations, Sherlock is able to detect the presence of the Illusion attack even with the execution of a single watchpoint.

Finally, we tested Sherlock with real workloads to measure its overhead upon the guest operating system's applications (Table 3). We carried out experiments with disk-, network-, and CPU-bound workloads. In our disk I/O experiment, we copied the 278 MB kernel source code tree from one directory to another using

Operations	*Normal VM (sec)*	*Sherlock (sec)*	*% Overhead*
Disk I/O	45.731	49.902	9.12
Network I/O (Virtual Network)	29.005	29.608	2.07
Network I/O (Physical Network)	212.054	213.352	0.61
CPU Bound	102.004	103.383	0.01

Table 3. Performance measurements. "Normal VM" indicates Xen without Sherlock monitoring; "Sherlock" includes monitoring time.

Linux's `cp` command. To test Sherlock against network workloads, we performed two different experiments. In the first experiment, we transferred a file of size 200 MB over HTTP between virtual machines using Xen's virtual network. We used a small HTTP server, `thttpd`, to serve the file. We repeated the file transfer operation on the same file using the physical network to a nearby machine. We measured the time taken by these file transfers with and without Sherlock and show results in Table 3.

Finally, we performed a CPU bound operation. Since we mainly instrumented I/O-related system calls (both disk and network), we did not expect significant overhead with CPU workloads. To measure CPU-bound cost, we used the `bzip2` utility to compress a tar archive of the Linux source tree. The result of this operation is again shown in Table 3. These results show that Sherlock is indeed an adaptive system that creates small overhead during the execution of benign applications.

As expected during the execution of these benign applications, Sherlock did not report any hidden operations.

5.3 Discussion

Sherlock relies on watchpoints to monitor the execution behavior of the kernel. In its current implementation, it requires the kernel's source code to calculate the watchpoints placement. This requirement makes it difficult to be used with closed source operating systems such as Windows. To address this problem, we plan to adopt an alternative design that uses driver isolation. In this new design, Sherlock would isolate the untrusted drivers in a different address space separate from the core kernel and prevent drivers entering into the kernel at arbitrary points; kernel code invocation would only be allowed through the exported kernel functions. This design is implemented by Srivastava et al. [38] to log the behavior of untrusted kernel drivers. With this design, we could correlate system calls with the kernel code execution behavior without requiring the source code or watchpoints.

In an extreme scenario, an attacker may bring the entire malicious functionality inside the malicious driver, copy the existing kernel code and remove watchpoints, or bring program slices of code for specific system calls to avoid hitting watchpoints inserted in the existing code. Though this kind of attacks are possible, it is difficult to launch as it requires the prediction of all possible

configurations of hardware and file systems present on the victim systems. These kind of attacks can be defeated by monitoring the untrusted driver's interaction with the hardware by using techniques similar to BitVisor [36] to know what operations are performed by the driver.

6 Conclusions

In this paper, we first presented an attack called *Illusion* that obfuscates the system-call executed by a malicious program. As a consequence, existing system-call based analyzers are not able to see the real operations performed by malicious code protected by the Illusion attack. Second, we presented a novel detection system named *Sherlock*. Sherlock detects the presence of the Illusion attack and exposes the hidden operations using a set of watchpoints inserted inside the kernel code of the guest OS. Thanks to its adaptive design, Sherlock is able to achieve its goal maintaining an acceptable performance overhead.

Acknowledgment of Support and Disclaimer We would like to thank our anonymous reviewers for their extremely helpful comments to improve the final version of the paper. We also thank Neha Sood for her comments on the early drafts of the paper. This material is based upon work supported by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or the U.S. Government.

References

1. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *12th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2005.
2. A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, May 2007.
3. Blorge. Faulty drivers bypass Vistas kernel protection. `http://vista.blorge.com/2007/08/02/faulty-drivers-bypass-vistas-kernel-protection/`. Last accessed 15 Jan 2011.
4. M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. In *Technical Report CMU-CS-02-197, Carnegie Mellon University, Pittsburg*, Dec. 2002.
5. F. David, E. Chan, J. Carlyle, and R. Campbell. Cloaker: hardware supported rootkit concealment. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
6. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *15th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.
7. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, May 1996.

8. V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux security modules framework. In *12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, Virginia, Nov. 2005.
9. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2004.
10. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
11. X. Jiang and X. Wang. Out-of-the-box monitoring of VM-based high-interaction honeypots. In *10th Recent Advances in Intrusion Detection (RAID)*, Sept. 2007.
12. S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference*, June 2006.
13. K. Kasslin. Kernel malware: The attack from within. `http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf`. Last accessed 15 Jan 2011.
14. M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Symposium on Operating System Principles (SOSP)*, Oct. 2007.
15. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, Aug. 2005.
16. J. V. Last. Stuxnet versus the iranian nuclear program. `http://www.sfexaminer.com/opinion/op-eds/2010/12/stuxnet-versus-iranian-nuclear-program`. Last accessed 15 Jan 2011.
17. C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *14th USENIX Security Symposium*, Aug. 2005.
18. L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2007.
19. L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.
20. A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Linux Symposium*, July 2006.
21. McAfee Security. System call interception. `http://www.crswann.com/3-NetworkSupport/SystemCall-Iinterception(McAfee).pdf`. Last accessed 15 Jan 2011.
22. D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.
23. K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *ACM Symposium on Applied Computing*, Mar. 2008.
24. packetstormsecurity. Adore rootkit. `http://packetstormsecurity.org/files/view/29692/adore-0.42.tgz`. Last accessed 15 Jan 2011.
25. packetstormsecurity. Knark rootkit. `http://packetstormsecurity.org/files/view/24853/knark-2.4.3.tgz`. Last accessed 15 Jan 2011.
26. B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, May 2008.

27. PCNews. Verisign working to mitigate stuxnet digital signature theft. `http://pcnews.uni.cc/verisign-working-to-mitigate-stuxnet-digital-signature-theft.html`. Last accessed 15 Jan 2011.
28. N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, Aug. 2006.
29. N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)*, Nov. 2007.
30. N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, Aug. 2003.
31. R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.
32. Rootkit.com. Rootkit.com. `http://www.rootkit.com/`. Last accessed 15 Jan 2011.
33. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, May 2001.
34. A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.
35. M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection: Formal analysis and practical models. In *10th Recent Advances in Intrusion Detection (RAID)*, Sept. 2007.
36. T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A thin hypervisor for enforcing I/O device security. In *ACM VEE*, Washington, DC, Mar. 2009.
37. Some Observations on Rootkits. Microsoft Malware Protection Center. http://blogs.technet.com/b/mmpc/archive/2010/01/07/some-observations-on-rootkits.aspx. Last accessed 15 Jan 2011.
38. A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In *RAID*, Ottawa, Canada, Sept. 2010.
39. A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *NDSS*, San Diego, California, Feb. 2011.
40. Sun Microsystem. Dtrace. `http://wikis.sun.com/display/DTrace/DTrace`. Last accessed 15 Jan 2011.
41. K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.
42. L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *USENIX Security Symposium*, Aug. 2008.
43. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, Nov. 2002.
44. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM CCS*, Chicago, IL, Nov. 2009.
45. H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *RAID*, Sept. 2004.
46. M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *ACM SACMAT*, June 2007.