Bridging the Semantic Gap to Mitigate Kernel-level Keyloggers

Jesus Navarro Enrique Naudon Daniela Oliveira
Computer Science Department
Bowdoin College, Brunswick ME USA
{jnavarro, enaudon, doliveir}@bowdoin.edu

***Abstract*—Kernel-level keyloggers, which are installed as part of the operating system (OS) with complete control of kernel code, data and resources, are a growing and very serious threat to the security of current systems. Defending against this type of malware means defending the kernel itself against compromise and it is still an open and difficult problem. This paper details the implementation of two classical kernel-level keyloggers for Linux 2.6.38 and how current defense approaches still fail to protect OSes against this type of malware. We further present our current research directions to mitigate this threat by employing an architecture where a guest OS and a virtual machine layer actively collaborate to guarantee kernel integrity. This collaborative approach allows us to better bridge the semantic gap between the OS and architecture layers and devise stronger and more flexible defense solutions to protect the integrity of OS kernels.**

I. Introduction

Kernel-level keyloggers are a growing and very serious threat to the security of current systems. A keylogger is a type of privacy-invasive malware that records keys pressed by a user. This data is usually saved as a file or sent directly to the network to third parties. The goal is to capture highly sensitive information such as passwords, user IDs, social security numbers, social media, email and gaming credentials that can be leaked for financial gain. A keylogger can record any stream of text and, depending on the machine being targeted, can also be used as a tool for espionage.

There are two types of keyloggers: user-level, which operates in unprivileged mode as a user-level process, and kernel-level (the focus of this paper), which is very dangerous as it operates as part of the operating system (OS) kernel with complete control of kernel code, data and resources. Defending against kernel-level keyloggers means defending the kernel itself against compromise, which is a difficult and open problem. Preventing, detecting and recovering from kernel attacks is difficult given the complexity of kernel code and the great number and variety of its data structures. This complexity makes it harder to determine known kernel good states usually employed in defense approaches. Attacks in the kernel can succeed not only by adding or changing kernel code or altering its control flow, but also by tampering with certain key non-control data structures and employing legitimate kernel code to perform malicious actions [29]. Further, an attacker has several avenues to compromise kernel integrity: vulnerabilities in kernel code, abuse of interfaces such as /dev/kmem [6] and malicious loadable kernel modules (LKM). The current solutions involving protecting the integrity of kernel are ineffective against attacks that do not rely on compromising existing kernel code or data structures and are able to leverage existing kernel code or APIs. We argue that protecting the OS kernel against kernel-level keyloggers is more difficult than protecting the kernel against general rootkits due to the keylogger's very simple model: eavesdrop and leak keystrokes.

This research was supported by the National Science Foundation grant CNS-1149730.

In this paper we discuss the implementation of two classical kernel-level keyloggers for Linux (kernel 2.6.38) based on [7], [1] and how they can defeat current approaches. The first kernel keylogger, an improvement of the keylogger described in [7] targets the terminal. In Linux, many important operations are done through the terminal, including operations that require an administrator or root password. If a root password is obtained, a malicious user would not have to implement any exploits to gain root access; he/she must simply log in like any normal user. The keylogger hijacks the *receive_buf* function from the tty line discipline. The second keylogger, based on [1] attempts to be as indistinguishable from legitimate kernel modules as possible. To this end, rather than proactively try to trick the kernel into giving it keystrokes by modifying specific kernel data structures or hijacking key kernel functions, it abuses a legitimate kernel API. Specifically, the second keylogger registers itself with the Linux keyboard notifier chain to capture keystrokes. Abusing the keyboard notifier chain allows it to successfully capture keystrokes without modifying any kernel code or data whatsoever. Furthermore, we can completely avoid violating any kernel invariants in the process. This "hide in plain sight" approach makes this keylogger resistant to security solutions that rely on checksums, and those that rely on monitoring kernel control flow, hooks or abuse of data structures.

Even though these attacks stress the vulnerability that current OSes face when confronted with such attacks, we believe that such threats can be mitigated. We also discuss in this paper a research direction we have been currently pursuing to prevent and detect kernel compromise: the employment of explicit collaboration between a guest OS and virtual machine (VM). By bridging the semantic gap between the architecture and system layers more fine-grained and stronger kernel keylogger protection mechanisms can be developed. Our contributions are as follows: (i) we discuss the implementation, features and limitations of two classes of kernel-level keyloggers for the Linux kernel (version 2.6.38), (ii) we discuss the limitations of current approaches facing these two keyloggers, and (iii) we present our research directions to mitigate such attacks by employing explicit collaboration between a guest OS and a VM layer.

The rest of the paper is organized as follows: Section 2 discusses related work, the current state-of-the-art in kernel integrity defense and the limitations of current defense approaches. Section 3 discusses the implementation of a keylogger that works by abusing the tty line discipline, while section 4 discusses a keylogger that "hides in plain sight" by

leveraging an existing kernel API. Section 5 addresses our research directions to mitigate such attacks and protect the kernel. We conclude this paper in section 6.

II. Related Work

The majority of existing keyloggers operates as a process in user-space and is installed in a computer usually bundled together with other types of malware like Trojan horses. They leverage sets of user-level APIs available in most OSes [35]. They can, for example, register themselves as keyboard listeners or constantly pool the state of the keyboard [25]. Many works in the literature have addressed their detection. These solutions usually rely on signatures, modeling of malware behavior based on system calls and library calls, and correlation between sensitive input to a system and I/O operations or memory writes in the kernel [10], [22], [26], [8], [35], [36]. As our keyloggers operate at kernel-level we will focus our discussion of related work on approaches that attempt to prevent or detect a compromise in the kernel, as any kernel-level malware will ultimately need to corrupt some kernel code or state to succeed.

A. State-of-the-Art in Kernel Defense

There is currently a vast body of research addressing kernel protection. Many works focusing on prevention use some form of code attestation or policy to decide whether or not a piece of code can be executed in kernel mode. Manitou [33] leverages a hypervisor to ensure that only authorized code runs in the system. It uses per-page permission bits to ensure that any code contained in an executable page is authorized for execution. Code is authenticated by taking cryptographic hashes of the page content before execution. SecVisor [43] employs a hypervisor to ensure that only user-approved code executes in kernel mode: users supply a policy, which is checked against all code loaded into the kernel. It virtualizes physical memory allowing hardware protections to be set over kernel memory. NICKLE [41] uses a memory shadowing scheme to prevent unauthorized code from executing in kernel mode. A trusted virtual machine monitor (VMM) maintains a shadow copy of the main memory for a running VM and performs kernel code authentication so that only trusted code is copied to the shadow memory. During execution, instructions are fetched only from the shadow memory. Code attestation techniques [23], [42] verify a piece of code before it gets loaded into the system. All these approaches do not protect kernel integrity against memory injection attacks. The main issue of having each code to be executed in kernel mode approved by a user administrator or a system policy is that this decision process is usually controlled by humans (a system administrator) and people make mistakes or can be deceived by social engineering techniques. In the lifetime of a system many drivers or modules must be installed anyway and these approaches burden the system administrator with the responsibility of determining which module is malicious or benign. Also they do not protect kernel pointers (hooks) from being subverted to compromise kernel control flow [45].

Also, few of these prevention approaches can guarantee some protection against non-control data attacks [19], for instance attacks that tamper with kernel data structures by directly injecting values into kernel memory (through vulnerabilities or by abusing /dev/kmem). Most solutions addressing prevention of non-control data attacks on kernel data structures rely on policies that explicitly consider the data structures to be protected. Given the great number and variety of them, these policies could be incomplete and fail to address all range of attacks. Livewire [24] is a VM architecture whose policies protect certain parts of kernel space such as the code section and the system call table from being modified. Paladin [13] relies on an administrator to specify access control policies for certain memory areas and system files. These policies protect files from being replaced and memory areas from being overwritten by using a containment algorithm based on a dependency tree of processes and files. Xu *et al.* [47] proposes a framework with an access control model for the specification of policies and an architecture to enforce kernel integrity. KernelGuard [40] prevents some dynamic data rootkit attacks by monitoring writes to selected data structures.

In previous work [34] we developed a proof-of-concept prototype for a system where a guest OS and a VM layer communicated to prevent tampering against kernel code and data segments at the architectural level. The system employed a dynamic information flow tracking (DIFT) system [9] that tainted network bytes. In a DIFT system we tag data (*e.g.*, bytes) with some extra information (*e.g.*, integrity or trust level) and track how this data flows throughout the system. The OS and VM exchanged information about the integrity (trust) level of objects at both levels of abstraction. Whenever a low integrity instruction attempted to perform a write operation into high integrity areas of kernel code or data the write was prevented (stopped) and a violation was signaled through an exception. It was successful against several control and non-control rootkits but it cannot defeat attacks that abuse existing kernel code and APIs.

HookSafe [45] is based on static and dynamic analysis and protects kernel hooks from being hijacked by relocating them to a dedicated memory space and regulating their access via hardware-based page-level protection. As it relies on dynamic analysis, it may be incomplete and could miss some hook access points. Secure Virtual Architecture (SVA) [20] is a virtual low-level typed instruction set that enforces a safe execution environment (memory safety, control-flow, type safety and sound analysis) for kernel code and its applications. It does not prevent, however, malicious code not exploring memory safety errors (buffer overflows, format strings, double frees) from corrupting the kernel and changing its behavior.

There are also many works addressing detection. Copilot [37] is a kernel integrity monitor that uses a PCI add-in card to access memory instead of relying on the kernel to accomplish that. The authors addressed Copilot's limitation of not being applicable to dynamic kernel data structures with an architecture that detects kernel violations by comparing the kernel state with a specification of a correct state done by an expert [38]. In a follow-up work, the authors also presented a technique that employs an approximation of control-flow graphs to periodically validate kernel state (each dynamically-computed branch is validated) during execution [32]. These techniques are not effective against attacks that tamper with kernel data structures (such as the resource wastage attack described by Arati *et al.* [14]) or that modify the kernel for short periods of time.

Strider GhostBuster [44], Lycosid [31] and VMWatcher [30] perform detection based on a cross-view approach: hiding behavior is captured by comparing two snapshots of the same state at the same time but from two different points of view (one from the malware and the other not). Gibraltar [12] detects rootkits modifying both control and non-control data. During a training phase the kernel execution is observed and

invariants for kernel data structures are hypothesized. These invariants are used as specifications of the integrity of data structures during an enforcement phase. A violation in these invariants indicates the presence of rootkits. Wang *et al.* [46] use a rootkit detection program to discover hooks that could be used by kernel rootkits to avoid detection. The solution is based on instrumenting and recording control-flow transfer instructions. Carbone *et al.* [15] proposed a system to map dynamic kernel data in a memory snapshot to enable integrity checking. OSck [28] protects the kernel by detecting violation in its invariants and monitoring its control flow. It monitors control flow by enforcing type signature (return type, number and type of parameters) of target functions. It will fail to detect a violation if the malicious function has the same type signature as the hijacked function or the rootkit leverages legitimate kernel code.

Moreover, none of these current defense approaches are able to defend the kernel against return-oriented rootkits [29], which reuse existing code within the kernel to perform malicious computations.

III. The TTY Keylogger

In the first keylogger, the terminal prompt was chosen as the target of the attack. In Linux, many important operations are done through the terminal, including operations that require an administrator or root password. If a root password is obtained, a malicious user would not have to implement any exploits to gain root access; he/she must simply log in like any normal user. This keylogger is an improvement of the techniques presented in [7] and was implemented as a LKM with complete access to the kernel. It has been tested (together with the second keylogger) on Ubuntu 11.04 running Linux 2.6.38 in Oracle VM VirtualBox version 4.1.6.

A. The TTY

For the keylogger to work, we must first find the target terminal prompt. In the Linux kernel, the struct *tty_struct* (linux/tty.h) is used to represent a terminal prompt. TTYs, short for teletype, were used as I/O devices back when computers were just beginning to multitask. Today, ttys are no longer physical teletypes, but emulated in software, despite working in the same way [4]. The tty (Figure 1) has a buffer to store characters, an associated driver (though there is no actual physical device) and an associated device (non-physical). For this keylogger, we focus on the tty line discipline [4], which is a layer of the tty that lies between the tty itself and the driver. It provides an intermediate step between getting characters from the tty driver to the tty, and providing functions like translating raw characters. One important function is *receive_buf()*, which is called by the tty driver to send characters to the line discipline. In our keylogger, we replace this function with our own keylogging function.

When the module is first initialized, we need to open up the tty. However, in any given Linux system, there is not a single tty. A list of ttys can be found in */dev/*. In general, a tty will be a file with the format /dev/tty/X where X is some integer. One user may be using */dev/tty7* while another is using */dev/tty63*. In the case of an xterm window, a pseudo-terminal is created under /dev/pts/X where X is also an integer. One implementation could be to hijack the *sys_open* system call. The new function would detect if a tty is being opened, then replace the necessary function. However, many rootkit detection systems protect the system call table, so this option is not the best. Instead, we simply open */dev/tty* without any trailing integer, which is not tied to a specific tty; it represents the currently active tty, regardless of whether or not it is an xterm window or a regular root prompt.

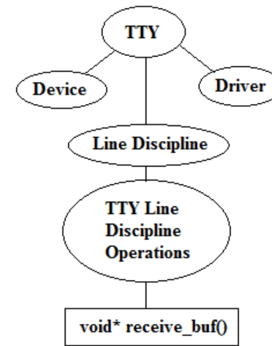

Fig. 1. The tty.

1) Getting the TTY: The first step is to open the */dev/tty* file and obtain the *tty_struct* from it. In Linux kernel, many entities, devices included, are treated as files, so we must open */dev/tty* as a file, represented by the *file* struct. We open the file using the *filp_open* function, which returns a pointer to the file. The *file* struct has a *private_data* field, which is of type (*void* *). In Linux, a tty is obtained by typecasting this void pointer to a *tty_file_private* pointer, a data structure specific to ttys. The *tty_file_private* struct has a field, which is a pointer to a tty. This pointer points to the *tty_struct* associated with the opened file.

2) Hijacking receive_buf: After obtaining the tty, we simply look for the *receive_buf* function, and if it exists, we replace it with our own. The *tty_struct* has a pointer to a *tty_ldisc* struct, which represents the line discipline of the tty. The *tty_ldisc* struct then points to a *tty_ldisc_ops* struct, short for tty line discipline operations. This struct contains function pointers to the functions provided by the line discipline, including the *receive_buf* function. The pointer chain looks like this: *tty->ldisc->ops->receive_buf*. With this function pointer, we first store the original function pointer, then replace the function with a pointer to our own function. Upon exiting the module, we restore the function pointer with the original pointer that we stored.

The *receive_buf* function does not return a value and takes the following parameters: *(struct tty_struct* tty, const unsigned char* cp, char* fp, int count)*. Our own function must match these parameters, even if it does not use them all, thus defeating OSck [28] rule of type safe, which requires the target function to have the same number and type of parameters of the original function in an uncompromised kernel. The parameters we are interested in are the *const unsigned char* cp*, which points to the char buffer that we want to log, and *int count*, which is the number of bytes contained in the buffer. In general, our new function writes these characters to our own buffer, and then writes our own buffer to a log file whenever the user presses the enter key.

B. Logging the Keys

When logging the keys, we cannot simply copy the keys as is. This is because there may be special keys pressed that cannot be printed easily, such as TAB or CTRL+C. The first thing our log function does is to check the count parameter. In most cases, this will be 1, and so we can expect an ASCII key.

The function will obtain this key and check for cases in which the user presses a special key. Rather than simply logging the key, we log a legible sequence of characters, such as *ˆC* (if the user pressed CTRL+C). These new characters and strings are stored in our own buffer until they are ready to be written to a log file. If the user presses a backspace key, we backtrack in our buffer, writing 0s into the keys that are deleted. When we find CTRL+M or \n (the user pressed enter), then we log it to our buffer and reset it, making it ready for a new set of keystrokes. We must also log keys when the tty is in canonical mode and not in raw mode. When in raw mode, the tty will not process the keys, giving us raw data instead [5], so we only obtain the keys after the tty has processed them.

We can also infer when the tty is expecting a password because it turns off echoing in this situation. The kernel provides us with macros to check this. When the user presses enter and the tty has echoing turned off, we add a message to our log file, indicating that the line that was just logged was a password. After the logging has been done, we call the original *receive_buf* function so that execution of the kernel can continue as usual.

C. Limitations and Defenses

This keylogger records only local terminal sessions. If, for example, the user remotely logs onto another host using *ssh*, those keys will not be logged. However, the user still has to remotely log into the host on a local terminal, so the username and password will still get logged. The keylogger is also limited to what keys it can log. This current implementation will only log single ASCII characters, but improving it to log other types of characters is straightforward. Because the *receive_buf* function indicates the number of chars/bytes in the buffer, counts greater than 1 will have special keys, which can be checked and logged appropriately. Another limitation is that, because */dev/tty* refers to the currently active tty, the keylogger does not distinguish between different ttys. A user could scramble our log file by typing between multiple terminal prompts in random order. This can be remedied as the *receive_buf* function passes a pointer to a tty which we could use to identify separate ttys and log them separately.

The best defense against this type of keylogger is to blacklist it. If it gets installed in the system by an administrator by mistake, bad judgment or social engineering deception most of current approaches will not be able to stop it. Copilot and its follow-up approaches [37], [38], [32] only perform checks for periods of time. HookSafe [45] cannot guarantee it will find and protect the *receive_buffer* hook, although it can include it as part of a policy. OSck [28] will treat our malicious function as benign as it has the same signature as the benign one. The strategy presented in [34] can defeat this keylogger as the hijack of the original *receive_buffer* function will correspond to a write into the kernel data segment by a suspicious instruction (its bytes came from the network).

IV. The Notifier-Chain Keylogger

This keylogger is based on [1] and, to remain hidden, it attempts to be as indistinguishable from legitimate kernel modules as possible. To this end, it abuses a legitimate kernel API rather than proactively try to trick the kernel into giving us keystrokes by modifying specific kernel data structures or hijacking key kernel functions. Specifically, it registers itself with the Linux's keyboard notifier chain to capture keystrokes.

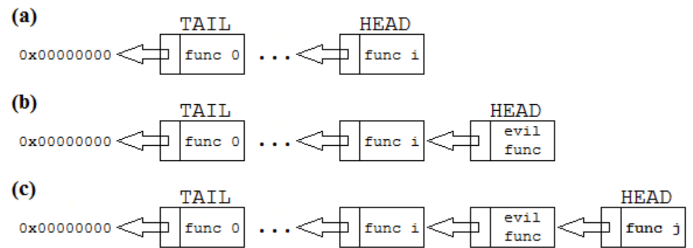

Fig. 2. Linux notifier chains.

Abusing the keyboard notifier chain allows this keylogger to successfully capture keystrokes without modifying any kernel code or data whatsoever. Furthermore, we can completely avoid violating any kernel invariants in the process. This "hide in plain sight" approach makes it much more resistant to security solutions that rely on checksums, monitoring kernel control flow and protecting function pointers.

A. Linux Notifier Chains

Notifier chains are a lightweight mechanism that facilitates flexible communication between portions of kernel code. They allow LKMs to request notification from other parts of the kernel when specific system events occur. This mechanism was initially developed in Linux v.1.1 to provide interested kernel modules with information about network events [3]. In subsequent kernel releases, more notifier chains have been added to the kernel, allowing code to request notifications about a wider range of possible events. A notifier chain is essentially just a linked list of callback functions. The kernel code responsible for detecting a given event must maintain the list head. When the given event occurs, the detecting module traverses the list, calling each function in the chain.

To register with a specific notifier chain, a module must add an entry to the head of the list with a pointer to its callback function. Any module may register a callback function with a notifier chain. Once a module no longer needs to be notified about a particular event, it simply removes its callback function from the chain. This strategy for publishing notifications about asynchronous events makes notifier chains extremely versatile, and well suited to handling communication between kernel modules. Figure 2 shows a malicious module adding a callback function to a notifier chain. Initially, the tail of the notifier chain is *func 0* and the head is *func i* (a). The tail's next pointer contains the value 0x00000000, or null, indicating the end of the list. When the malicious module adds its callback function, *evil func*, to the notifier chain, it is added to the head of the list. The next pointer of *evil func* points to *func i*, now the second function in the list (b). As other modules register callback functions with the notifier chain, the head of the list is updated. For example, when *func j* is added, it becomes the list's new head (c).

In Linux v.2.6.24, a keyboard notifier chain was added, allowing interested modules to request notification whenever a keystroke occurs. To capture keystroke information, our second keylogger registers a malicious callback function with the keyboard notifier chain, as illustrated by Figure 2. When a keystroke occurs, the notifier chain calls the callback function, and passes a pointer to a *keyboard_notifier_param* struct (Figure 3), which contains information about the keystroke.

B. Translation

The *keyboard_notifier_param* structure can provide information about which key was pressed in three different formats:

```
struct keyboard_notifier_param {
      ...
      int down;              //key-press or key-release
      int shift;             //shift mask
      int ledstate;          //led state
      unsigned int value;    //keycode, unicode or keysym
};
```

Fig. 3. The keyboard_notifier_param struct.

keycodes, Unicode or key symbols. While Unicode information is more convenient, as it can be logged directly, it is only used for special function keys that come on certain types of keyboards, such as media keys (play, pause, etc.) and web navigation keys (home, forward, backward, etc.) [2]. We could use keycode or key symbol information to log keystrokes, but key symbols are more convenient. Keycodes are 7-bit quantities that uniquely identify each key on a keyboard [2]. While this is convenient because it would allow our keylogger to distinguish between any two keys on the keyboard, it becomes tedious because duplicate keys (left and right ctrl, alt and shift keys, for example) generally should be treated identically in our log file.

Key symbols are 16-bit quantities. The high order byte provides type information, while the low order byte provides value information [2]. To translate key symbols into ASCII characters, the Linux kernel uses the high byte to select the appropriate translation function. The translation function then uses the value to return an ASCII representation of the keystroke. For example, the key symbols 0xf103 and 0xf303 both have a value of 0x03, but they correspond to different keys (the F4 key, and the number pad's 3 key, respectively). Using type information, the kernel selects different translation functions to handle these two key symbols. We mimic this strategy, but define our own translation functions. Writing a custom translation functions grants us greater freedom in determining how our log file looks.

C. Limitations and Defenses

This keylogger cannot log keystrokes from *ssh* or other remote login sessions. This is because keystrokes from remote sessions come in over network interfaces, and the keyboard notifier chain only detects keystrokes generated by the physical keyboard. This keylogger does not corrupt any kernel hook, data structure or any portion of kernel code and data segments. It can evade all previously discussed defense approaches if not blacklisted from installation in the first place.

V. Kernel Protection

The problem of defending against malware that can modify the OS remains open and is difficult. The majority of the proposed solutions for protecting the kernel against compromises employ virtualization. In the traditional VM usage model [18], it is assumed that the VM is trustworthy and the OS running on top of it can be easily compromised by malware. This traditional usage model comes with a cost: the semantic gap problem (Figure 4). There is significant difference between the abstractions or the state observed by the guest OS (high level semantic information) and by the VM (lower level semantic information). Current VM-based kernel defense solutions do not count on active collaboration between a guest OS and a VM to enhance system security or bridge this semantic gap. They use a technique called introspection to extract meaningful information from the system they monitor/protect [24].

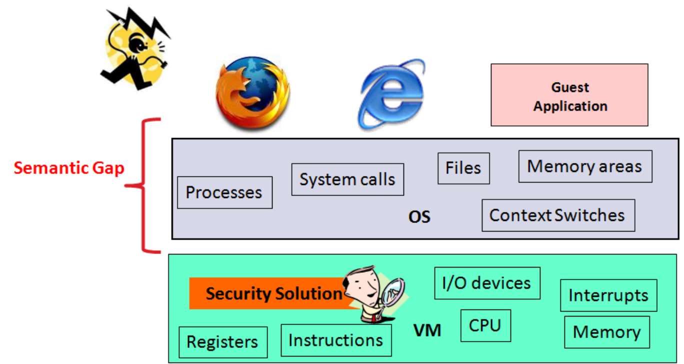

Fig. 4. The semantic gap.

With introspection, the physical memory of the current VM instance is inspected and high level information is obtained by using detailed knowledge of the OS algorithms and data structures [21], *i.e.*, the OS state is inspected from an outside entity, a VMM. Introspection solutions assume that even if the guest OS is compromised, their mechanisms and tools, residing at a lower-level (VM), will continue to report accurate results. However, as argued by Baram *et al.* [11], these solutions share the questionable assumption that the original kernel data structures and memory layout are not tampered with by the untrusted guest OS they want to protect. This assumption does not hold because kernel-level malware can indeed tamper with kernel data structures (structure, semantic, location) so as three views of the system are provided [11]: (i) an external view, which is bogus and is intended for an introspection/defense tool to see, (ii) an internal view which is bogus and is intended for the guest OS to see and, (iii) the actual view, known only to the attacker. Thus, introspection-based defense tools do end up relying on the integrity of the guest OS to function correctly. They depend on the guest OS data structures and algorithms to be uncompromised to report correct results and such assumption does not hold in the face of kernel-level malware that directly manipulates these data [11]. This raises the following question: why not leverage the guest OS in VM-based security approaches?

We are currently researching a novel architecture that challenges this traditional model. Our proposal is to improve introspection between a guest OS and VM by employing the concept of collaboration. This allows the VM to get a much better idea of what is going on in the guest OS without having to reverse engineer it from low-level data structures. Collaboration helps closing this semantic gap, thus allowing for stronger and more flexible security approaches to be developed. We claim that (i) this paradigm allows for more fine-grained and flexible security solutions to be developed and, (ii) this approach is no less secure than the traditional model, as introspection tools also depend on the kernel data structures and algorithms to be untampered to report correct results [11].

In this architecture the OS downcalls the VM through a protected new software interrupt that will only be used for this purpose, *i.e.*, it will be an unused software interrupt in the Intel x86 ISA (Instruction Set Architecture). This software interrupt will be only allowed to be invoked from within selected kernel functions and we will only permit legitimate kernel code inside a trusted execution context to invoke it (a system call, for example). This prevents malicious return-

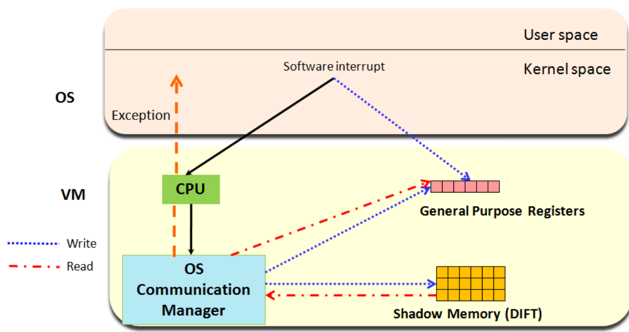

Fig. 5. The collaborative architecture.

oriented malware [29], built from random instructions from the kernel but not belonging to a trusted execution context to initiate bogus communications with the VM. For example, the VM could check if the instructions actually belong to a particular system call, function or handler instead of an instruction from a random memory location even if it belongs to the kernel. The VM processes the OS downcall and returns information in general purpose registers or in special protected OS memory areas. If the VM initiates a communication with the OS it does so using new exceptions (Figure 5).

Why is collaboration better to help bridge this semantic gap? By actively involving the guest OS into the virtualization layer we are able to combine two key advantages: (i) the best possible view of objects at OS level, and (ii) the isolation and hardware extensibility properties of VMs.

Our assumption is that the VM is trustworthy and the guest OS might be partially compromised. We say partially because we assume that the interface between guest OS and VM is assumed to be untampered. This interface and associated memory locations, as we discussed above, are protected at the architecture level. This is feasible because they represent a much smaller trusted computing base when compared to the entire OS kernel.

The architecture envisioned is DIFT-based [9]. DIFT (dynamic information flow tracking system) is a powerful and promising technique with several different applications in security solutions. Currently two challenges prevent it from being widely deployed in protection schemes: performance and stability, which prevents these systems from tracking all necessary dependencies. Fortunately, current research have been addressing these issues [39], [27], [17], [16], [9]. In spite of that, this architecture will be designed so that it can be employed in a non-DIFT manner as well. The DIFT system is associated with the VM and is responsible for marking instruction bytes with a certain integrity level based on a general policy, *e.g.*, all bytes coming from network source X are considered low-integrity.

To defeat kernel-level keyloggers we will distinguish instructions performed by different subjects at kernel level. For example, when the CPU is executing a particular instruction the VM needs to know whether or not: (i) the instruction is executing at kernel or user level, (ii) the OS is in interrupt or process context, or executing instructions from a kernel thread or module, (iii) the instruction is from original kernel code or new code (LKM) installed from the network, (iv) the module where instruction comes from is trusted, (v) the instruction is performing a read operation from a sensitive memory area. The main idea is to mark keyboard data at the source as sensitive and keep track of how this data is being used at the VM layer, *e.g.*which operation is being performed (read/write), privilege level of the instruction executing and which execution context. Based on this information, the VM should decide whether or not to allow the operation (*e.g.*a read). If the operation is not to be allowed the VM should not only prevent it but also notify the OS about this violation. The system may decide to stop executing instructions from that particular context (*e.g.*, a module) altogether, uninstall the module, or just continue executing the instructions but restrict the module's access to system resources and kernel areas. In this strategy, the OS is responsible for instructing the VM in advance about boundaries of sensitive memory areas and data structures. For example, to defeat the first keylogger the OS can instruct the VM to mark the keylogger buffer *char *cp* memory region at architecture level as sensitive and the VM would not allow read operations from suspicious instructions, for example, modules installed from the network. For the second keylogger, the OS could instruct the VM to mark the keyboard notifier chain data structure as sensitive at architecture level and only allow execution of non-suspicious callback functions. This approach allows modules to be installed in the system (relieving the burden of blacklisting) and execute normally as long as they do not attempt to perform any suspicious operation, *e.g.*, read from a keyboard buffer.

VI. Conclusions

Despite the great number of kernel defense approaches proposed in the literature against keyloggers or other types of kernel-level malware the problem is still open and difficult. OS kernels have a very complex design in general with a great number and variety of data structures, hooks and many other avenues for exploitation. New generation of malware are starting to explore existing kernel code and APIs to succeed. In this paper we described the implementation of two kernel-level keyloggers and stressed the need for novel, stronger and more flexible defense approaches. Explicit collaboration between a guest OS and a VM layer underneath it to bridge the semantic gap between these two layers of abstraction, as sketched in this paper, seems like a promising research direction to better protect the OS kernel integrity.

References

[1] How to: Building your own kernel space keylogger. (http://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html).

[2] The linux keyboard driver. (http://www.linuxjournal.com/article/1080?page=0).

[3] Notifier chains and completion functions. (http://www.linux-mag.com/id/2711/).

[4] The tty demystified. (http://www.linusakesson.net/programming/tty/index.php).

[5] The tty in raw mode. http://uw714doc.sco.com/en/SDK_sysprog/_TTY_in_Raw_Mode.html.

[6] Who needs /dev/kmem? (http://lwn.net/Articles/147901/).

[7] Writing linux kernel keylogger. phrack magazine http://www.phrack.org/issues.html?issue=59&id=14.

[8] Y. Al-Hammadi and U. Aickelin. Detecting Bots Based on Keylogging Activities . *International Conference on Availability, Reliability and Security*, pages 896–902, 2008.

[9] M. I. Al-Saleh and J. R. Crandall. On information flow for intrusion detection: What if accurate full-system dynamic information flow tracking was possible? *New Security Paradigms Workshop*, 2010.

[10] M. Aslam, R. N. Idrees, M. M. Baig, and M. A. Arshad. Anti-Hook Shield against the Software Key Loggers . *National Conference on Emerging Technologies*, 2004.

[11] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 82–91, 2010.

[12] A. Baliga, V. Ganapathy, and L. Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. *Annual Computer Security Applications Conference (ACSAC)*, pages 77–86, December 2008.

[13] A. Baliga and L. Iftode. Automated Containment of Rootkit Attacks. *Computer and Security, Elsevier*, 2008.

[14] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. *IEEE S&P'07*, pages 246–251, May 2007.

[15] M. Carbone, W. Lee, W. Cui, M. Peinado, L. Lu, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. *ACM CCS*, 2009.

[16] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. *ACM CCS*, November 2008.

[17] H. Chen, X. Wu, L. Yuan, B. Zang, P. chung Yew, and F. T. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. *ISCA*, June 2008.

[18] P. M. Chen and B. D. Noble. When Virtual is Better than Real. *HotOS*, May 2001.

[19] S. Chen, J. Xu, and E. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security*, 2005.

[20] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. *SOSP*, October 2007.

[21] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. *IEEE Security and Privacy*, 2011.

[22] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis . *USENIX Annual Technical Conference*, pages 233–246, 2007.

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[24] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Network and Distributed System Security Symposium*, 2003.

[25] M. T. Goodrich and R. Tamassia. *Introduction to Computer Security*. Addison Wesley, 2011.

[26] J. Han, J. Kwon, and H. Lee. HoneyID: Unveiling Hidden Spywares by Generating Bogus Events. *IFIP International Information Security Conference*, 278:669–673, 2008.

[27] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *EuroSys*, 2006.

[28] O. S. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring Operating System Kernel Integrity with OSck. *ASPLOS*, 2011.

[29] R. Hund, T. Holz, and F. C. Freiling. Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. *USENIX Security*, 2009.

[30] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. *ACM CCS*, pages 128–138, November 2007.

[31] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[32] N. L. P. Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. *ACM CCS*, November 2007.

[33] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. *ASID*, October 2006.

[34] D. Oliveira and S. F. Wu. Protecting Kernel Code and Data with a Virtualization-Aware Collaborative Operating System. *Annual Computer Security Applications Conference (ACSAC)*, December 2009.

[35] S. Ortolani, C. Giuffrida, and B. Crispo. Bait your hook: a novel detection technique for keyloggers. *Conference on Recent Advances in Intrusion Detection (RAID)*, 2010.

[36] S. Ortolani, C. Giuffrida, and B. Crispo. KLIMAX: Profiling Memory Writes to Detect Keystroke-Harvesting Malware. *RAID*, 2011.

[37] N. Petroni, T. Fraser, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *USENIX*, 2004.

[38] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. *USENIX Security*, 2006.

[39] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. *MICRO-39*, pages 135–148, December 2006.

[40] 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*, 2009.

[41] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. *RAID*, 2008.

[42] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. *ACM CCS*, November 2004.

[43] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[44] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. *DSN*, 2005.

[45] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. *ACM CCS*, 2009.

[46] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. *RAID*, September 2008.

[47] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. *SACMAT*, 2007.