Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring

Junghwan Rhee, Ryan Riley, Dongyan Xu
CERIAS and Department of Computer Science
Purdue University
West Lafayette, IN, 47907
{rhee,rileyrd,dxu}@cs.purdue.edu

Xuxian Jiang
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
jiang@csc.ncsu.edu

***Abstract*—Targeting the operating system kernel, the core of trust in a system, kernel rootkits are able to compromise the entire system, placing it under malicious control, while eluding detection efforts. Within the realm of kernel rootkits, *dynamic data rootkits* are particularly elusive due to the fact that they attack only data targets. Dynamic data rootkits avoid code injection and instead use existing kernel code to manipulate kernel data. Because they do not execute any new code, they are able to complete their attacks without violating kernel code integrity.**

We propose a prevention solution that blocks dynamic data kernel rootkit attacks by monitoring kernel memory access using virtual machine monitor (VMM) policies. Although the VMM is an external monitor, our system preemptively detects changes to monitored kernel data states and enables fine-grained inspection of memory accesses on dynamically changing kernel data. In addition, readable and writable kernel data can be protected by exposing the illegal use of existing code by dynamic data kernel rootkits.

We have implemented a prototype of our system using the QEMU VMM. Our experiments show that it successfully defeats synthesized dynamic data kernel rootkits in real-time, demonstrating its effectiveness and practicality.

I. Introduction

Kernel rootkits [1]–[4] are one of the most technically challenging malware to defend against because of their attack targets: operating system (OS) kernels. The OS kernel is the core of trust in a running system, with every program relying on the information and environment it provides.

As such, if the kernel is compromised then there is effectively no trusted component remaining within the victim system. This leaves all the system's software, including security software, vulnerable to being deceived by forged kernel and system states. Rootkits have the goal of taking control of a victim OS while eluding intrusion detection systems. A common technique is to hijack kernel control flow or manipulate kernel states in order to effectively blind the system and prevent detection of the intrusion. Recent attack trends [5], [6] show that kernel rootkits are increasingly being used to make other malware more effective.

This paper focuses on the prevention of kernel *dynamic data* attacks which may be employed by a rootkit. A traditional kernel rootkit attack involves the injection and execution of malicious code at the kernel level. A dynamic data attack, however, operates by directly modifying dynamically allocated kernel data structures without the use of injected code. These data structures may validly be located at changing addresses, contain variant data, or have a changing number of runtime instantiations. Modern rootkit detection and prevention systems are unable to effectively counter this type of attack.

Currently, there are two major approaches to that are used to approach the problem of rootkit defense. Both approaches require an *external* monitoring system in order to ensure a high resistance to attacks from inside the system. (In essence, any effective rootkit defense system must run at a privilege level higher than that of the rootkit.) The first type of approach works to guarantee the integrity of executed kernel code [7], [8]. This prevention technique is able to defeat kernel rootkits that require the execution of unauthorized code in the kernel's address space. Dynamic data attacks, however, do not require the victim kernel to execute any new, unauthorized kernel code and as such are able to freely bypass code integrity based approaches. The second approach uses passive monitors [9], [10] to obtain snapshots of kernel memory and analyzes them to detect rootkit attacks based on kernel control flow integrity violations. This technique detects violations based on changes to invariant kernel content, however attacks against data with variant content make such approaches simply unapplicable. In addition, because this method relies on the periodic capturing of snapshots, it cannot capture or prevent the *action* of intrusion, instead it can only detect the intrusion after the attack has taken place. This *passive* nature makes such an approach vulnerable to dynamic data attacks which are coordinated to be launched and withdrawn between snapshot periods. There are other, noticeably less effective, approaches that place the monitor inside the system to be protected [11]–[17]. None of these approaches can detect the manipulation of data using existing kernel code, therefore dynamic data kernel attacks bypass these approaches as well.

In this work, we present a system that aims at *preventing* dynamic data kernel attacks. Our system uses VMM-based memory access monitoring to *actively prevent* malicious memory accesses on protected kernel data transparently without any changes to the guest operating system. Protecting kernel data is a significant challenge due to its dynamic nature. For example, the location, content, or even the number of instances of various pieces of kernel data are all subject to change. Using

an external monitor (such as a VMM) to fully track the state changes of dynamic kernel data within the monitored system is a non-trivial task. The contribution of this paper is a state-of-the-art technique able to prevent attacks on kernel data. In addition, advanced VM introspection techniques useful for tracking dynamic properties of protected data in real-time are also contributed. We also present KernelGuard (KG), which is an instantiation of our techniques using the QEMU VMM. Our experiments show that KG successfully defeats synthesized dynamic data kernel rootkit attacks.

The rest of this paper is organized as follows: Section II introduces dynamic data kernel rootkits. Section III describes the dynamic characteristics of kernel data and the challenges in applying VMM monitoring techniques to them. The KG implementation is shown in section IV. We evaluate the effectiveness and performance of KG in section V. Section VI has discussions on our techniques and section VII presents related work. Finally we conclude this paper in section VIII.

II. Dynamic Data Kernel Rootkit Attacks

In this section we introduce a set of dynamic data rootkits with advanced, elusive properties. First, let us define this type of rootkit.

Definition: Dynamic data kernel rootkits are a family of kernel rootkits which target kernel data that has dynamic characteristics (e.g. variant content, variable location, or a dynamic number of runtime instantiations) without injecting any code into the kernel memory space.

These rootkits belong to the Direct Kernel Object Manipulation (DKOM) [18] family of attacks which target kernel data. They are a specific subset of that family, however, that avoids the use of any malicious code injection or loading code (e.g., loadable kernel modules [19]) into the kernel space. The use of malicious code run with the kernel's privilege level may be used as a fingerprint of the attack by detection or prevention mechanisms [7], [8]. Instead, these rootkits accomplish their attacks using direct memory access devices. In Linux, `/dev/kmem` and `/dev/mem` are the notorious examples of such devices and have been used in the past by kernel rootkits. The phrack article on SucKIT [20] is a well-known introduction to `/dev/kmem` attacks. SucKIT, however, only used `/dev/kmem` to inject malicious code and hijack the system call table. As such, it was detectable by conventional rootkit defense systems. However, when the general technique demonstrated by SucKIT is used to attack kernel data directly without injecting any code, we can implement new attacks which have advanced, evading characteristics. Although we present work implemented on Linux, direct kernel memory manipulation is a general problem on operating systems with similar memory devices. In Windows, `\Device\PhysicalMemory` and `\Device\DebugMemory` are the equivalent devices. In both operating systems the original intent of these devices is for efficient access to video memory, kernel debugging, or memory forensics. However, the unrestricted capability to read and write kernel memory allows malicious kernel rootkits to read any content or dynamically change the kernel to alter its behavior or state.

A. Threat Model

Our threat model considers an attacker who has root privileges on a given Linux system and desires to mask his presence. He is permitted to use the direct memory access devices (such as `/dev/kmem`) but he will not inject or otherwise execute any new code with the kernel's privilege level. (We operate under the assumption that execution of new code will be detected or prevented immediately using existing work in the area.) This means he may modify any and all data in the kernel as well as potentially make use of existing kernel code. The kernel data includes most kernel data structures including kernel function pointers. (However, he will not use those to attempt to execute new code.) The main techniques of dynamic data attacks include changing the value of data, changing the structure of data (e.g., by manipulating pointers), or changing the control flow without violating code integrity. We present examples of these attack types in the following section.

B. Attack Examples and Effects

Given an attacker that is not permitted to execute new code, it seems prudent to ask the question, "How much damage can such a limited attacker really cause?" Work by Petroni et al. [10] found that 96% of the real world Linux rootkits they analyzed required persistent changes to the kernel's control flow. The attacker presented in this work, however, is severely limited in his ability to make such changes. Can rootkits still cause significant damage despite this limitation? In this section, we demonstrate several types dynamic data attacks available for use by rootkits. All of these attacks have the ability to effectively carry out an attacker's goals without the execution of new code. The possible attack vectors for dynamic data attacks are not limited to the cases presented here. In later sections, we will discuss the challenges of dynamic data attacks and specific defense mechanisms available to defeat them.

Anti-Security Mechanism - Shutting Down SELinux Manipulating only a kernel data structure can have a substantial impact by actively disarming a security system. For example, an attacker could disable a major security system, SELinux, using the `/dev/kmem` device. We installed SELinux based on Linux Security Modules (LSM) on top of Redhat 8 and performed an experiment to disable the policy enforcement of the SELinux kernel. The `selinux_ops` structure contains pointers to all of the SELinux auditing functions. Whenever a permission check is needed, the kernel uses this table to determine which function to call. This table is registered to be pointed to by the generic security module pointer, `security_ops`. To disable SELinux's policy enforcement mechanism, we patched `security_ops` at runtime to point to a dummy initialization structure that simply bypasses all checks. This effectively bypasses and disarms the enforcement mechanisms of SELinux.

Resource Status Manipulation - The Lying Network Card Carefully manipulating kernel resource reports can create a false view of resource consumption capable of evading intrusion detection systems (IDS) which monitor resources. Consider the case where an attacker is running a hidden malicious service that makes significant use of the network. Network usage status (e.g., number of bytes transmitted or received) would provide obvious evidence of an attacker's presence. Using a dynamic data attack we can manipulate the in-kernel status variables for the network card interface and hence cause `ifconfig` and other network applications to display forged statistics.

Hidden Services - Process Hiding One important goal of rootkits is to ensure that any malicious processes are not visible to the system administrator or IDS. Botnets, for example, may use rootkits to hide the malicious bot program that is running. This simple act of process hiding can be accomplished using a dynamic data attack. We can manipulate the task list structure using the `/dev/kmem` device, effectively removing the malicious task from view, in order to conceal the running service. This attack can be further enhanced by using it together with the preceding resource status manipulation to create the illusion of a normal system both in terms of running processes and resource consumption.

C. Characteristics of Dynamic data Attacks

There are several characteristics of dynamic data kernel rootkits that allow them to elude existing defense systems.

No Code Integrity Violation One of the most effective mechanisms for defending against kernel rootkits is to protect or verify the kernel code integrity of the operating system [7], [8]. This approach defends against kernel rootkits that inject malicious code by ensuring that only approved code can execute in kernel mode. Dynamic data kernel rootkits, however, target variant kernel structures which are set with read and write permissions, and the attack is performed by using already approved kernel code which satisfies kernel code integrity. Dynamic data attacks are not covered by kernel code integrity checking.

Dynamic states Some methods of rootkit defense operate by analyzing memory dumps of the kernel's memory and comparing key portions of it to known good values [9]. For example, the detector may ensure that the system call table has not been manipulated. This technique, however, is not effective for protecting portions of the kernel that are permitted to change and do not have invariant good values. The usage status of a network card, for example, is represented by kernel variables that are constantly updated at runtime. A static memory dump of the data, even taken periodically, does not reveal whether or not the values have been changed. In addition, changes to the variables are common and cannot be used as an indicator of the presence of a rootkit. As such, dynamic data rootkits targeting dynamic kernel data are able to circumvent standard integrity checking because the the validity of the data is not obvious.

Transient Transient attacks are another way dynamic data rootkits can be more robust against monitoring. A state-based control flow integrity (SBCFI) monitor [10] may passively detect a subset of dynamic data attacks if the possible states of data are enumerable and a policy is used to decide desired or undesired states. However, such a monitor analyzes periodic snapshots, therefore the system is still vulnerable if the attack is well coordinated to do its damage between snapshots. Being asynchronous, SBCFI monitor does not monitor every state change that occurs in the system, thus it is feasible for an attack to remain undetected if it occurs in its entirety between two monitoring periods.

III. Design and Techniques

Our approach is designed to be totally transparent to the operating system (OS) being protected, meaning that we require no changes to the underlying OS. In order to transparently support the OS, we monitor the execution of the OS at the instruction level within the VMM. While we operate at the instruction level, in order to defeat dynamic data attacks we need to have a higher level view of the kernel's memory. We start with a high level abstraction of what to protect and how to protect it and use that information to enable protection at the lower level where KG runs.

From a high level, the system functions as follows. For each kernel data structure that needs to be protected, a policy is written which describes how the VMM should identify the data structure in a raw view of memory as well as the characteristics of an attack against that data structure. In addition, the policy describes the pointers within the kernel's memory that point to the data structure so that those can be tracked and protected as well. At runtime, the VMM finds the data structure in memory and intercepts all writes to its address in order to validate them and ensure they do not violate the policy. In addition, the pointers specified are also monitored in order to ensure that if the kernel moves the data structure in memory KG can still monitor it. This ensures that dynamic data structures can still be monitored in real time, even if they change locations. In the event a malicious write is detected, it can be prevented at the KG level.

In this section, we first present the assumptions of our design. Then we show the main challenges in defeating dynamic data kernel attacks as well as the unique design and techniques used to overcome such challenges.

A. Assumptions

In our design we assume that the monitor is activated after the operating system has booted and reached a stable state. This is because the kernel data structures undergo significant changes during system initialization, and it is not important to track those changes until they are completed. As an example, consider the case where we want to monitor and protect the data structures for the network card. During boot up that particular data structure does not exist until the card is initialized, and as such it would not be fruitful to attempt to monitor it before that. As a consequence of this startup assumption, we

also assume that the system reaches an initialized state before the attacker begins his attack.

B. Dynamic Characteristics of Kernel Data

Before we describe how we protect dynamic kernel data, we note that there is a significant challenge in determining, at runtime, exactly where that dynamic kernel data is located in memory. Unlike kernel code or static data, dynamic kernel data may exist in different portions of memory at various times and may move at runtime. Due to the fact that one of the key mechanisms KG uses is to track every access to a protected data structure, it is important to know exactly where that structure is at all times. If the monitor has a stale address then it can either fail to detect an attack or falsely prevent valid memory accesses. This means that as dynamic data structures are moved or the number of instantiations of them change, KG must track those changes.

In this section, we describe the characteristics of dynamic kernel data that make it difficult for the monitor to maintain a correct list of memory addresses.

Location of Data Where data is located in the kernel's memory plays a large role in determining the best way to protect it. We classify data location into two categories.

- *Static (Immobile)*: Data has a fixed location determined at compile time. (e.g., `init_task_union` is a statically declared task structure for the first process.)
- *Dynamic (Mobile)*: Location is decided at runtime and subject to change. (e.g., task structures for new processes created during the runtime of the system.)

Content of Data If the value of a data item can vary, the content cannot always be used to determine whether manipulation has occurred. We classify data content into two categories.

- *Static (Invariant)*: Data has the constant value that should never change.
- *Dynamic (Variant)*: Data can have arbitrary or multiple possible values.

Number of Instances of Data A data structure may be composed of multiple instances of itself (e.g., array, a linked list) and if the number of instances changes, the VMM should capture that change. We classify the number of instances into two categories.

- *Static (Fixed)*: Data is known to have a fixed number of instances and addresses at compile time.
- *Dynamic (Changing)*: The number of data instances in the structure may change at runtime.

Many current rootkits target data that has a static location, invariant content and a fixed number of instances (e.g., kernel code, the system call table, or the interrupt descriptor table). More effective attacks, however, can target mobile data structures containing variant content and a changing number of instances (e.g., the process list). In the following section, we introduce techniques to handle such compilicated characteristics of dynamic data.

C. Dealing with the Dynamic Characteristics of Kernel Data

Due to the fact that the VMM exists outside of the kernel's execution state, it cannot easily access internal kernel data like the kernel does. Instead of having easy access to the process list, for example, the VMM must instead extract the list from a raw view of memory. This divide is referred to as the *semantic gap* [21]. The semantic gap is the difference between the high-level operating system context of a guest VM and the low-level hardware states available to the VMM. In order to protect kernel data, the operating system's higher-level context for it must be reconstructed.

Our system requires more than a simple bridging of the semantic gap. In order to ensure proper protection of dynamic data, (i.e., varying in size or location) the address of the data structure being protected must be kept up to date and reflect any valid changes the operating system may make. In order to accomplish this, KG uses two primitives designed to assist in watching memory. The first primitive, the *memoryguard*, is a general purpose protection mechanism to guard a data structure at a given range of memory addresses. Any memory write to the memory range protected by a memoryguard is intercepted and validated in order to determine if it constitutes an attack. Its role is analogous to the protection offered by a page table that simply applies a page's permissions to memory accesses on the page. A memoryguard, however, is unique in that it can apply policies to those accesses in order to distinguish between legal and illegal access attempts. The second primitive we have established is a *watchpoint* which assists in protecting data by actively detecting pointer state changes that should cause memoryguards to be updated. A watchpoint watches memory accesses to a pointer to the data structure protected by a memoryguard. Therefore, if the protected data moves (which is reflected by a change in the pointer), the watchpoint detects the address change of the moving data and triggers an update to the associated memoryguard to reflect the address change.

In this section we describe the techniques employed by KG to deal with the dynamic characteristics of kernel data. The techniques are able to overcome the semantic gap to allow the VMM to capture and track changes to dynamic kernel data in real time.

Location of Data Depending on the location of the data being protected, different methods are used to determine the raw memory addresses to be guarded.

- *Static*: The address of the data is known at compile time and can be found from the kernel symbol table (i.e., `System.map` in Linux). This is easy to determine and does not require any sort of tracking to watch for the data to move within memory.
- *Dynamic*: The most difficult type of data addresses to determine and track are for dynamic, moving kernel data. Dynamic kernel data is declared as a pointer somewhere in memory to a dynamically allocated portion of memory. At runtime, the pointer may be updated to point to a new portion of dynamic memory and the old data structure

is removed. The VMM would not normally be aware of this type of change. It is important when protecting dynamic data structures that the VMM knows the current address of the structure lest it find itself guarding an old version. We overcame the difficulty in tracking dynamic data by using the watch primitives to assist in monitoring the pointer and the data it points to. The memoryguard is used by KG to track the data structure itself. The watchpoint is used to watch pointers to the data protected by memoryguards and update it if need be.

Content of Data Now that we know how to track the locations of data structures that we want to protect, it is important to describe how we actually protect the data structures themselves. This task is difficult because KG must distinguish between valid and invalid changes to the kernel data structures.

- *Static*: Since invariant data should not change, protecting it is a simple matter of ensuring that it does not change. Traditional content integrity checking approaches evaluate the content of invariant data structures and signal an alert if they are changed. We take a more active approach by preemptively preventing writes to invariant data structures.
- *Dynamic*: When content is allowed to vary (sometimes arbitrarily) one cannot simply deny writes to it in order to protect it, nor can one necessarily determine the validity of the write based on its content. As such, a new technique is needed. For KG we have developed a way to detect illegal memory accesses using an *Access Invariant* property. The property is described in detail in section III-D, however the main idea is that for every piece of variant data we want to protect we explicitly enumerate the kernel functions allowed to or prohibited from operating on it. This technique is instantiated in the memoryguards, as mentioned earlier in this section.

Number of Instances of Data When KG is activated, all protected data items are individually tracked and marked to be protected. While static data can be effectively tracked by marking it just once during KG's activation, dynamic data requires continuous monitoring since the number of instances of an item can change as the data structure is updated at runtime.

- *Static*: When KG is activated, the addresses of individual instances are discovered by traversing the kernel data structures from within the VMM.
- *Dynamic*: To detect any changes to the monitored data structures, watchpoints are created using pointers that signify the structural changes. Therefore any restructuring will trigger the watchpoint and the whole data structure will be re-identified.

D. Kernel Memory Access Monitoring

Once the data to be protected is found and tracked using memoryguards and watchpoints, KG will intercede during any memory writes to those memory ranges. This section presents our methodology for determining whether a given write to a protected kernel data structure is valid or instead the result of a rootkit attack.

Input: A write address (addr), CPU instruction pointer (EIP)

```
1:  if the update is scheduled then
2:     UpdateWatchPrimitives(GetScheduledTag())
3:  end if
4:  (prot, tag) ⇐ IsProtected (addr)
5:  if prot = NotProtected then
6:     {allow the memory write on unprotected memory}
7:  else
8:     if IsProhibitedCode (addr, EIP) then
9:        print "Code EIP attacks the target at addr."
10:       {prevent the memory write to addr by EIP}
11:    else
12:       if prot = WatchPoint then
13:          schedule the update of watch primitives with tag
14:       end if
15:       {allow the benign memory write}
16:    end if
17: end if
```

Fig. 1. An algorithm to verify a kernel memory write

Access Invariants: Protecting Variant Content As mentioned before, dynamic data rootkits evade code integrity based monitors by only attacking data, usually using a direct memory access device. An important insight to realize about such attacks is that sort of access does not exist in the original kernel code. Based on the source code, we can determine which functions are normally used to access a given kernel data structure and which functions should never be used to access a given kernel data structure. By detecting when a data structure is being modified from an unauthorized function, we can detect and prevent rootkit attacks. We call the property that the data should be accessed only by the functions intended in the original kernel context an *Access Invariant*. KG explicitly enforces this property by describing what kernel code is allowed to or prohibited from accessing protected kernel data. It exposes dynamic data attacks by determining when protected data is manipulated by kernel functions that are not intended to do so. The legitimacy of memory access is validated by referring to the program counter (i.e., EIP register) when a memory access is intercepted and inspected for an illegal access.

E. Memory Watch Primitives and Their Dynamic Management

As presented in section III-C, our system uses memory watch primitives to protect kernel data and track the movement of protected data within the guest machine. Internally a watch primitive is interpreted as a condition of memory ranges whereby the VMM intercepts all memory accesses within the guest machine.

Figure 1 shows the algorithm for integrated operation of watchpoint monitoring and updating of monitored data structures in the VMM's context. When a write occurs to kernel code or data, the address of that write is checked to determine

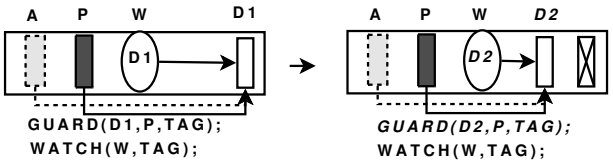

Fig. 2. The address change of protected data from D1 to D2 is recognized by the corresponding watchpoint and the watch primitives (i.e. memoryguard and watchpoint) are updated using the tag, TAG. (address ranges: `A` for allowed access function, `P` for prohibited access function, `W` for a watchpoint, and `D1`, `D2` for moving, protected data)

whether or not it corresponds to any of the currently active watch primitives.

The function `IsProtected(addr)` queries an efficient data structure similar to a page table in order to determine the protection information for the write target. If the target address is protected, the memory address is further examined by querying the program counter in order to determine if the instruction issuing the writes is permitted to do so, and if not concludes that an attack is occuring. If the write is deemed permissible, then it assumes the write is either a valid access (for a memoryguard) or a signal to update watch primitives due to the movement of protected data (for a watchpoint). Memory writes which do not meet any of the conditions mentioned above are allowed to proceed.

Figure 2 illustrates how the watchpoint update signal is processed. In this Figure, the VMM is using a watchpoint and a memoryguard to protect a data structure at the initial address range `D1`. The watchpoint for the pointer at the address range `W` tracks dynamic address changes of the protected data from `D1` to `D2`. First, a watchpoint detects a change to the pointer it is tracking because it is changed from `D1` to `D2`. (This signifies that a memoryguard needs to be updated.) Second, the existing memoryguard that governs the protection of the previous data structure is flushed using the tag, `TAG`, associated with the watchpoint. Third, the removed memoryguard is regenerated by the VMM for the new address of the data, `D2`. This process generates the updated memoryguard by the description of the policy that shares the same tag with the triggered watchpoint.

IV. Implementation

The methodology described in section III is portable to any VMM that supports memory interposition (e.g., Bochs, QEMU, VirtualBox, or VMware Workstation). Memory interposition is generally a part of VMMs which translate guest instructions into host instructions. VMMs that do not support memory interposition (e.g., Xen or VMWare ESX) can also support this technique using traps on the data pages [7], [22]. For our prototype implementation, we chose to use the QEMU virtual machine platform to implement KG. QEMU uses a dynamic recompiler to translate guest instructions into host instructions. Instead of constantly recompiling all guest code, QEMU maintains a cache of recently recompiled and used blocks of code in order to improve performance. We inserted our interception routines into this cache code in order to implement interposition effectively and efficiently.

A. Exposing and Preventing Dynamic data Attacks

One specific instance of a dynamic data attack writes its payload from user space using direct memory access devices such as `/dev/kmem` or `/dev/mem`. The devices use a kernel function, `__generic_copy_from_user`, to actually copy data from user space into its new location within the kernel. This is a general function used by various parts of the kernel to copy data from user space. Based on the analysis of kernel source code, however, we determined that in normal usage this function should never be used to overwrite our protected kernel data structures. As such, we can make use of the Access Invariant property to realize that any writes to our protected data structures using this function are malicious. This means that we can effectively prevent this type of dynamic data rootkit with a simple policy for prohibited functions specifying that `__generic_copy_from_user` is prohibited from writing to our protected data structures. Note that this does not disable proper usage of the direct memory access devices, but instead prevents them from overwriting protected kernel data structures, including dynamic ones.

B. Optimizing Monitoring Performance

Due to the fact that our approach only needs to operate on kernel data being modified by kernel code, write interception does not need to occur during user mode in order for our technique to be functional. (This is because user mode code does not have the proper privileges to write to kernel memory.) This permits us to leverage the KQEMU acceleration module for QEMU. KQEMU is able to execute user mode code directly on the host processor (noticeably faster than standard QEMU) while still executing kernel mode code using the KG instrumented QEMU recompiler. This results in noticeable performance improvements.

V. Evaluation

In this section we will describe the experiments run to validate the effectiveness and practicality of our defense system. In order to test KG against dynamic data attacks we implemented and tested the three dynamic data attacks described in section II-B. We also ran five performance benchmark programs to ascertain the performance overhead associated with kernel memory access monitoring and maintaining a synchronous view of the guest VM's data structures.

For the SELinux attack experiment, we used a kernel built using the LSM-based SELinux implementation patch applied to a vanilla 2.4.21 Linux kernel. For the other two attack experiments as well as the performance tests, the OS of the guest virtual machine is an off-the-shelf copy of Redhat 8 with an unmodified copy of version 2.4.18 of the Linux kernel. We used QEMU version 0.9.0. A guest system is configured with 512MB RAM, and a host system has a 3.2GHz Pentium D CPU and 2GB RAM. QEMU does not support dual CPUs, therefore one core was effectively used for experiments.

A. Preventing Dynamic Data Kernel Rootkit Attacks

In order to protect against the three attacks described in section II-B we created policies applicable to them. In each policy, we used the kernel semantic information that is extracted from kernel and the VMM action that generates watch primitives. For all three policies we used the address range of the `__generic_copy_from_user` kernel function to create a memoryguard which prevents the attack from using the direct memory access devices to write to the data structure being protected.

Protecting SELinux The attack shown in section II-B turns off policy enforcement in SELinux by patching the general security module pointer `security_ops`. The protection of this pointer is sufficient to prevent the attack. The `security_ops` pointer is statically declared, therefore its address is determined at compile time. The policy prevents write access on the `security_ops` from all direct memory access devices.

Preventing NIC Data Manipulation In Linux, an individual NIC is represented by a `net_device` structure and all of a system's NICs are organized in a linked list pointed to by a global static pointer, `dev_base`. NIC status information is stored in a separate structure, `net_device_stats`. This is the structure we desire to protect. For a loopback device the status structure is accessible using `*(net_device.priv)`. For other network devices, `*(net_device.priv).stat` should be used. The policy tells the VMM how to traverse the list of NIC structures and generate memoryguards to protect the status data.

Preventing Process Hiding The process list consists of multiple `task_struct` objects which can be traversed by following the `next_task` or `prev_task` pointers. The head of the list, `init_task_union`, is statically declared, therefore its address is determined at compile time. The VMM traverses the list using kernel semantic information and generates a memoryguard and a watchpoint for each object. All watch primitives for `task_struct` objects share the same tag, `TAG_PROC`, so that when the list is modified due to the addition or removal of an object the list can be updated using one tag.

B. Runtime Performance

During performance testing, we loaded the system with the policies to prevent NIC manipulation and process hiding. The policy for SELinux protection is not used in this performance section because we intended to use the off-the-shelf guest OS for performance measurement while the use of SELinux requires a custom compilation of the kernel.

In order to evaluate the efficiency of KG, we compared the performance of our implementation with the performance of an unmodified QEMU virtual machine. Both KG and QEMU ran with KQEMU support in normal mode which causes QEMU to execute user mode instructions directly on the host CPU. We chose five applications as benchmarks and compared the results of running them with and without the protection.

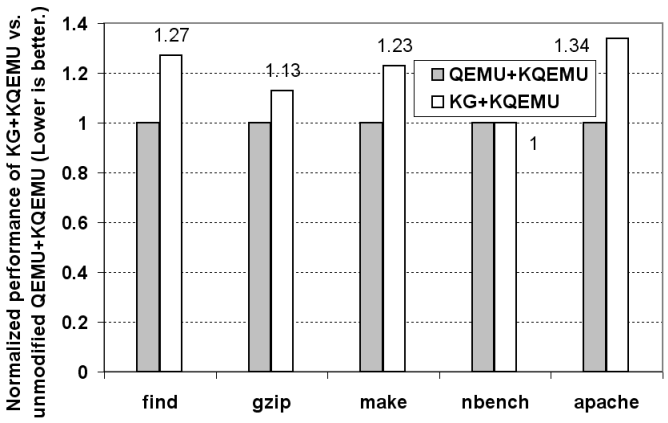

Fig. 3. Performance results of KG with KQEMU support compared to unmodified QEMU with KQEMU support. Results are normalized with respect to the unmodified VMM. In the apache benchmark, time per request is used. In the other benchmarks, the total execution time is used.

All benchmarks except `apache` were measured in terms of execution time. In the `apache` benchmark the average process time per request was used for comparison instead. The `find` benchmark shows a workload with intensive disk and filesystem activity. The `gzip` benchmark compresses the archived source code of the Linux 2.4.18-14 kernel. This benchmark has a workload of intensive user level arithmetic and filesystem access. The `make` benchmark shows compilation of version 2.4.18-14 of the Linux kernel. It tests the performance of the kernel development tool chain (e.g., gcc, ld and make) and involves both a CPU workload and intensive I/O operations. The `nbench` benchmark runs the Linux/Unix nbench 2.2.2 benchmarking program. It measures the performance of the CPU, FPU and memory system. It is primarily CPU bound. For the `apache` benchmark we ran Apache version 2.2.8 within the guest VM and used the ApacheBench tool from outside of the VM to determine its throughput. We configured ApacheBench to use 3 concurrent connections and 20000 requests for each connection.

Figure 3 shows the performance of KG memory access monitoring. As can be seen, the more memory intensive an application is, the higher the overhead. CPU bound applications, on the other hand, experience little or no slowdown.

VI. Discussion

Our system uses semantic information about each piece of data inside guest machine in order to monitor and protect them. Therefore, our approach is currently specification-based and can be applied to protect specific, core data structures which require security protection which is not provided by any existing solutions. The focus of this paper is to highlight the challenges of protecting against dynamic data rootkits and provide novel design techniques to overcome those difficulties. In order to expand the coverage of protected data, one could envision using static analysis of the kernel source code and/or dynamic analysis to capture dynamic runtime memory management information. We are currently pursuing these avenues

as future work.

Our solution is based on the access invariant concept which determines malicious attacks due to abnormal memory accesses occuring from locations that were not part of the original kernel's context. Therefore, an attack which made use of specific, existing kernel code in a return-to-libc style attack would be able to bypass parts of our system.

VII. Related Work

SecVisor [7] and NICKLE [8] detect kernel rootkits by enforcing kernel code integrity. Their approach is subset of the general concept of protecting *invariant content*. When the attack is directed toward dynamic data using existing kernel code, their approach cannot be applied to defeat the attack.

SBCFI [10] detects kernel rootkits based on permanent attack traces observed in periodic memory snapshots. Although this approach has extended protection coverage for function pointers, the protection is effective only for targets with invariant or enumerable content. In addition, taking an asynchronous approach may permit dynamic data attacks.

Specification-based approaches [23] determine attacks based on the violation of kernel semantic relationships held among data structures at runtime. However, this approach may not be applicable to guard data structures which do not have semantic relationships with others. In contrast, our approach can protect them because attacks are determined on the basis of illegal memory accesses on the data.

Livewire [24] is the work which introduced the VMM-based intrusion detection system and VM introspection technique. Their work does not cover advanced attacks such as those demonstrated in this paper.

Dynamic data kernel rootkits and user level non-control data attacks [25] share the property of avoiding direct control flow changes while still inflicting their damage. The difference is that kernel attacks require a higher privilege level to manipulate kernel data, and direct memory access devices are used for initiating attacks against the OS kernel. Two memory access monitoring techniques for user level programs [26], [27] have been proposed that use similar approaches to identify software vulnerabilities at the user level. Unlike these two techniques, our work can protect unmodified operating system kernels without any hardware architectural support or source code rewriting using VM introspection.

VIII. Conclusion

In this paper, we present a prevention system that defeats dynamic data kernel rootkit attacks. Our approach is based on memory access monitoring which requires the VMM to maintain the precise address of dynamic kernel data. We overcome this challenge by using memory watch primitives that instantly signal the state changes of data. The dynamic data attacks are exposed using the enforcement of the Access Invariant property to detect the unintended use of existing kernel code for malicious purposes. Our proof-of-concept implementation on the QEMU VMM is able to successfully defeat synthesized dynamic data kernel attacks based on the presented techniques.

Acknowledgment

This work was supported in part by NSF Grants CNS-0716376, CNS-0716444 and CNS-0546173.

References

[1] Stealth, *adore-0.42*. http://stealth.7350.org/rootkits.
[2] Stealth, "adore-ng-0.53," http://stealth.7350.org/rootkits.
[3] fuzen_op, "FU rootkit," https://www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip.
[4] P. Silverman and C.H.A.O.S., "FUTo," http://www.rootkit.com/newsread.php?newsid=433.
[5] Fanbot, "W32/Fanbot.A@mm," http://www.symantec.com/security_response/writeup.jsp?docid=2005-101715-5745-99.
[6] E. Florio, "When Malware meets Rootkits," http://www.symantec.com/avcenter/reference/when.malware.meets.rootkits.pdf.
[7] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ACM, 2007.
[8] Ryan Riley and Xuxian Jiang and Dongyan Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing," in *RAID 2008: Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.
[9] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings for the 13th USENIX Security Symposium*, August 2004.
[10] N. L. Petroni and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 103–115, ACM, 2007.
[11] M. Boelen, "Rootkit Hunter 1.2.9," http://rkhunter.sourceforge.net/.
[12] N. Murilo and K. Steding-Jessen, "chkrootkit V. 0.47," http://www.chkrootkit.org.
[13] F-Secure, "F-Secure Blacklight," http://www.f-secure.com/blacklight/blacklight.html.
[14] B. Cogswell and M. Russinovich, "RootkitRevealer v1.71," http://www.microsoft.com/technet/sysinternals/utilities/RootkitRevealer.mspx.
[15] pjf_, "IceSword 1.20," http://www.blogcn.com/user17/pjf/index.html.
[16] T. Lalwess, "Saint Jude," http://www.sourceforge.net/projects/stjude.
[17] Foundstone, "Carbonite - a rootkit detection and analyzer," http://www.foundstone.com/rdlabs/proddesc/carbonite.html.
[18] J. Butler, "DKOM (Direct Kernel Object Manipulation)," http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.
[19] K. J. Jones, "Loadable Kernel Modules," *;login: The Magazine of USENIX and SAGE, 26(7)*, November 2001.
[20] devik and sd, "Linux on-the-fly kernel patching without LKM," http://www.phrack.org/issues.html?id=7&issue=58.
[21] P. M. Chen and B. D. Noble, "When Virtual Is Better Than Real," in *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, IEEE Computer Society, 2001.
[22] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Oakland '08: Proceedings of 2008 IEEE Symposium on Security and Privacy*, (Oakland, CA, USA), IEEE Computer Society, 2008.
[23] 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," in *Proceedings for the 15th USENIX Security Symposium*, (Vancouver, B.C., Canada), July 2006.
[24] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, February 2003.
[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, 2005.
[26] P. Zhou and W. Liu and F. Long and S. Lu and F. Qin and Y. Zhou and S. Midkiff and J. Torrellas, "Accmon: Automatically detecting memory-related bugs via program counter-based invariants," in *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, 2004.
[27] Emre C. Sezer and Peng Ning and Chongkyung Kil and Jun Xu, "Memsherlock: an automated debugger for unknown memory corruption vulnerabilities," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, ACM, 2007.