DKSM: Subverting Virtual Machine Introspection for Fun and Profit

Sina Bahram, Xuxian Jiang, Zhi Wang
Mike Grace, Jinku Li, Deepa Srinivasan
North Carolina State University
{sbahram, xjiang4, zhi_wang, mcgrace}@ncsu.edu

Junghwan Rhee, Dongyan Xu
Department of Computer Science
Purdue University
{rhee, dxu}@cs.purdue.edu

Abstract

Virtual machine (VM) introspection is a powerful technique for determining the specific aspects of guest VM execution from outside the VM. Unfortunately, existing introspection solutions share a common questionable assumption. This assumption is embodied in the expectation that original kernel data structures are respected by the untrusted guest and thus can be directly used to bridge the well-known semantic gap. In this paper, we assume the perspective of the attacker, and exploit this questionable assumption to subvert VM introspection. In particular, we present an attack called DKSM *(Direct Kernel Structure Manipulation), and show that it can effectively foil existing VM introspection solutions into providing false information. By assuming this perspective, we hope to better understand the challenges and opportunities for the development of future reliable VM introspection solutions that are* not *vulnerable to the proposed attack.*

Keywords: *Virtualization, Introspection, Direct Kernel Structure Manipulation*

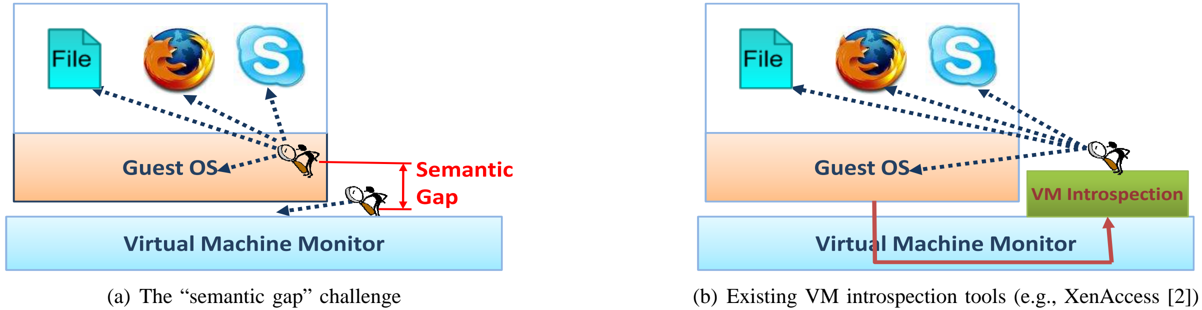
I. Introduction

Research in virtualization technologies has gained significant momentum in recent years, mainly due to the many new opportunities to address a variety of computer system problems (including security and reliability). One key technique behind these opportunities is called *virtual machine introspection*. The goal of VM introspection is to enable the observation of a VM's states and events from outside the VM. In particular, this outside observation can have the same (or similar) semantic view of system states and events as if they were seen from inside the VM. This observability is critical to enable tamper-resistant, high-fidelity VM monitoring, which is in turn the basis of a wide range of opportunities being actively explored, such as introspection-based intrusion detection, fault-tolerance, service hosting, and dynamic resource provisioning.

VM introspection has been touted as an extremely powerful technique and a number of recent systems have been successfully developed to demonstrate its great potential. For example, Livewire [11] is the first introspection-based intrusion detection system that aims to protect running guest VMs from being compromised (e.g., by kernel rootkits). XenAccess [2], VMwatcher [13], VMwall [23], and others [14], [15] were developed to monitor VM execution and infer guest-internal states or events (e.g., running processes, loaded kernel modules, active network connections, or ongoing guest system calls). These guest-internal states and events are needed for the purpose of either recording or denying the execution of suspicious programs. Most recently, VMware has introduced VMsafe [25] technology that can allow third-party security vendors to leverage the unique benefits of VM introspection to better monitor, protect, and control guest VMs.

The capability to introspect running VMs opens up many opportunities that are simply not possible with physical machines. A key challenge, however, that needs to be overcome is the so-called *semantic gap* [9] between the external and internal observations of a VM (Figure 1(a)). Specifically, from outside the VM, we can get a view of the VM at the virtual machine monitor (VMM) level, which includes its register values, memory pages, disk blocks and low-level events (e.g., execution of a privileged instruction); whereas from inside the VM, we can observe semantic-level entities (e.g. processes and files) and events (e.g., system calls). This semantic gap is formed by the vast difference between external and internal observations, manifesting the main challenge for VM introspection.

To bridge the semantic gap (Figure 1(b)), one key observation behind existing introspection tools is that the guest OS being introspected contains a set of data structures (e.g., those for process and file system management), which can be used as "templates" to interpret VMM-level VM observations. As such, we can cast low-level VM observations to guest OS data structures to uncover the VM's semantic entities and their states. As an example, by casting VM memory page content to guest OS data structure definitions, we can locate and identify kernel data structures that have been defined to maintain semantic



(a) The "semantic gap" challenge

(b) Existing VM introspection tools (e.g., XenAccess [2])

Fig. 1. Existing VM introspection tools bridge the "semantic gap"

entities in the VM (e.g., process control blocks and kernel modules). By following these data structures, we can further derive their attributes (e.g., a process' name, PID, page table etc.). In particular, for a process, we can explore its virtual address space through its page table and derive its user-level states (e.g., variables at specific memory locations).

However, a careful examination of existing VM introspection tools as well as our past experiences in building some of them [13] indicate that *the effectiveness and reliability of VM introspection does not naturally come without question*. In particular, the main concern stems from a common, fundamental assumption of VM introspection: the guest OS being introspected is assumed to use the kernel data in a prescribed fashion by following these data structure templates. In other words, all existing introspection tools rely upon the fact that the underlying guest OS is conforming to certain behaviors and idioms (with respect to these templates) which would, at first glance, appear to be rather obvious and set in stone. Unfortunately, as is most often the case with introspection, the guest OS kernel could be compromised. And once compromised, the assumption about the kernel respecting its own data structures becomes seriously questionable.

In this paper, we present *Direct Kernel Structure Manipulation (DKSM)*, an attack which can effectively subvert and confound existing VM introspection tools. Specifically, by presenting DKSM, we show that it is possible to compromise a guest such that the kernel's use of any field of its data structures (or templates) could be potentially modified. The modification can be achieved by various techniques that involve changing the associated syntax and semantics of the underlying data structures, thus invalidating the fundamental assumption of introspection. With this invalidation, we are effectively creating three different views of the system: (1) The first, *internal*, view comprises what the OS (or various system routines such as *ps* and *netstat*) sees; (2) The second, *external*, view comprises what an external introspection-based tool observes; (3) The third, *actual*, view comprises what is really going on in the system. A piece of malware (e.g., a kernel rootkit) that implements our proposed attack technique can effectively control all three of these views. This means that an attacker can present any desired *external* view of the system to evade or circumvent VM introspection, while presenting a completely different *internal* view of the system to the guest. However, neither of these two views will represent the *actual* view of what is executing in the system.

We have developed a proof-of-concept DKSM prototype to illustrate this attack and show the fragility of existing introspection tools. In our prototype, we have successfully misrepresented several important types of information to existing VM introspection tools. Specifically, by manipulating information about *running processes*, *loaded modules*, and *active network connections*, we show that it is possible for a new (and stealthier) class of introspection-resistant malware to emerge. In addition, we also examine possible strategies to better cloak the DKSM attack and "raise the bar" even further for future enhanced VM introspection techniques. Meanwhile, it is important to note that by effectively evading existing VM introspection tools, our goal here is to expose the fundamental limitation of these tools; thereby arguing for the need of next-generation VM introspection techniques with enhanced tamper-resistance. We believe this higher level of diligence is increasingly important especially considering the current trend of adopting virtualization and related introspection-capable applications.

II. DKSM Design

In a nutshell, DKSM foils existing introspection techniques by attacking the basic assumption upon which they are based. Specifically, it is important to notice that DKSM is based on the observation that the kernel data structures (or templates) of a guest VM are key to any introspection tool, which means a DKSM attack can be launched in various ways to manipulate these kernel data structures. In the following, we examine three different approaches: (1) *syntax-based manipulation* where certain fields of kernel data structures are potentially added or removed; (2) *semantics-based manipulation* where the semantics of the underlying data structures are changed; and (3) *multifaceted combo manipulation* which effectively combines the previous two. Note all these attacks can effectively manipulate the kernel data structures to subvert introspection tools and their analysis.

In this work, we assume the presence of guest kernel vulnerabilities that can be exploited by the attacker to compromise the guest kernel and hijack the control flow. We consider this adversarial model because this is often the case how an introspection tool is deployed. By hijacking control flow, the attacker has the freedom to either modify existing kernel code or inject his own code for execution. We also notice the recent emergence of return-oriented programming [7], [20], [12] and believe it is possible to launch a similarly return-oriented DKSM attack. This possibility will be discussed in Section III-C.

A. Syntax-based Manipulation

One potential approach to implementing a DKSM attack involves adding or removing specific fields from particular kernel data structures. By doing so, the template-based approach of existing introspection tools will be using the wrong templates to infer guest states and thus derive inaccurate results. Note that an added member field to the kernel data structure may not impact introspection as the analysis of existing member fields could be sufficient in inferring guest VM states. However, a removed member field could greatly affect the accuracy and reliability of the results. As mentioned in [10], certain fields of kernel data structures are simply not used by the OS.[1] As a result, their removal would not adversely affect the OS kernel behavior. When an introspection tool depends on any of these removed fields for the analysis, such syntax-based manipulation will evade or even mislead the tool analysis.

From another perspective, we notice this form of the DKSM attack is subject to certain limitations as it does not fundamentally hide or change the underlying semantics of the affected data structures. What happens is that the data structure types are syntactically manipulated. Therefore, if an introspection tool is changed to lessen its dependency upon the syntax of related templates (e.g., by leveraging only essential member fields), the chances are high that this attack is less effective.

B. Semantics-based Manipulation

A more advanced form of the DKSM attack is to modify the underlying semantics of the kernel data structures of interest. Because the changed semantics can be transparent to the OS while offering great consternation to introspection tools, we consider this approach to be much more powerful than the previous approach.

Specifically, kernel data structures contain a number of member fields. Often, some of these fields are of similar types or widths. For example, in a kernel data structure that contains several integer values, one interesting attack involves switching these integers around. By doing so, the underlying OS can still use the correct integer fields of the data structure. However, an external introspection tool would assume the standard layout of the data structure (before the members are switched) to be true; therefore, it would assign incorrect semantics to these switched member fields. Further, such a technique can be easily extended from integers to strings, and eventually support any data types. Here, we also note that homogeneity of data types is not required at all, which means that an integer field can be used to contain a pointer to a C-style string, while a previous C-style string pointer can be used as an actual integer. Effectively, such a technique hides the actual data in plain sight as it has been neither redirected nor changed in any way. However, when the data is being introspected, the template-based view precludes the examination of the actual way in which this data is used. As such, the introspection technique is unable to derive accurate semantics about the affected data structures.

From another perspective, we can also achieve semantics-based manipulation by redirecting chosen fields of kernel data structures to somewhere else. In other words, various fields of interest are redirected to shadow locations instead of being exchanged with other members. By doing so, the attack can create a data structure that is physically dis-contiguous (and the affected OS kernel will be instrumented to operate seamlessly with these dis-contiguous kernel objects), thus greatly impacting the template-based introspection view. One nice property of this approach is that introspection tools are still presented with a perfectly natural and normal looking view of the system. No fields are missing or added, nor do the meanings of various member fields in the kernel data structures appear to be unusual. Unfortunately, for the introspection tools, the OS is not using this data in any meaningful way. As a result, we can further use it to provide false information in an attempt to mislead or confuse the external view generated by introspection.

Specifically, a rootkit that chooses to employ this specific technique has the intriguing opportunity to present an external view of the system (to introspection tools), as well as an internal view (to running anti-virus software for example). The *external* view is false and consists of what the rootkit wishes the introspection tool to see. Similarly, the *internal* view can be anything and everything that the rootkit wishes the OS to see. Neither of these views have to be remotely accurate. They simply have to satisfy current expectations, such as those of the underlying OS, or fundamental assumptions, such as those of introspection. However, the reality, i.e., the *actual* view, of what is going on in the system can be hidden away from all parties.

[1] A similar observation has led to the examination of reliability of kernel data structure signatures for memory forensic analysis. Specifically, when a signature involves a member that is not being used by the OS kernel, an attacker could potentially misuse this member to evade or mislead signature-based analysis. One such example is the backward pointer in the all task list [10].

C. Multifaceted Combo Manipulation

A third approach is to combine the above two attacks together to launch a multifaceted combo attack. This is possible because there is no limitation in the presented attacks restricting them to be mutually exclusive. Specifically, a combo attack would involve first redirecting all accesses of the specific data, followed by switching the various sibling members with one another, while simultaneously adding and removing unused member fields to the various data structures. An introspection tool would have to first untangle the complex interweaved web of semantics encompassing each structure (since all the introspection tool's templates now have become useless). Then it would have to try to derive and understand the appropriate semantics of the various data structures. After it has successfully realized such a hard goal, the introspection tool is finally left with the reality that none of this data is even being used because it has all been redirected. We notice that this particular process is remarkably similar to the existing arms-race between malware obfuscation and reverse engineering. As a result, DKSM can make things arbitrarily hard for analysis.

This combo attack interestingly puts the attacker in the position of a "defender". One is, in essence, defending against the introspection tool's attempts to find out the truth. The various stages of the attack can be viewed as layers of "defense" that the introspection tool must break through to find the truth, only to be presented with another false and even more complicated set of circumstances to have to deal with. In addition, certain implementation details such as shadow scheme and return scheme (Section III) make such an attack very attractive as it can not only effectively subvert all existing introspection tools, but also significantly raise the bar for next-generation ones.

III. Implementation Strategies

In this section, we present different schemes with varying stealthiness guarantees and respective prototyping difficulties to implement the proposed DKSM attack. Specifically, we first examine a *direct scheme* that achieves the DKSM effect by directly changing existing kernel code that accesses the kernel data. Then we present a *shadow scheme* that shadows the execution of the attack code to achieve the same goal (e.g., by hijacking kernel control flow without tampering with existing kernel code). We then present a *return scheme* that misuses existing code via the use of return-oriented programming [20], [12]. Each scheme builds upon the previous one. In the following, we also examine the strengths and weaknesses of each scheme.

A. Direct Scheme

In the direct scheme, the kernel code which accesses the data is directly manipulated. Using an exploit, malicious LKM, or other delivery mechanism to gain root access, this scheme proceeds to overwrite those instructions that access data of interest and detour their execution. When those instructions are to be executed, the detoured execution will include additional logic to decide how those instructions should be instrumented to reflect the *internal* view of the current kernel data. For example, when certain member fields are re-located to another memory page, the corresponding kernel data-accessing code will be redirected to access the data from the new location. Also, when storing the redirected data (e.g., PIDs, port numbers, and module names), we can further obfuscate this data to foil potential analysis attempts. For instance, the data can be split apart such that PIDs do not appear close to one another in the redirected form, strings can be altered programmatically to foil statistical analysis, and so on.

We note that this scheme can be detected if a VM introspection tool chooses to examine the memory locations containing the original instructions and verify kernel code integrity. Specifically, a tool that carried out such an analysis would easily detect the compromise of kernel code. And by analyzing how the code is modified, the tool may piece together a real view of the system, independent of the various results of the syntax and semantics based attacks. Although the obfuscations of kernel data and non-contiguous layout of kernel objects can make this task significantly harder, a persistent introspection tool may eventually defeat these obfuscations. More importantly, the introspection tool would infer that the guest kernel code was compromised, thus exposing the attack.

B. Shadow Scheme

From another perspective, our second scheme aims to use a shadow memory implementation to increase the attack's stealthiness and thus make it harder for detection. Particularly, a shadow memory implementation is a specific form of split memory where data and code are separated from one another. Our scheme is inspired by available split memory systems [18], [19], [22], [24] on the *x86* architecture and misuses this technique for our attack. Specifically, in a shadow memory implementation, one can exploit the caching mechanisms of the *x86* architecture to present one view of memory mapped in cache, while the original code memory pages simply contain the untampered values to potentially mislead introspection.

Note this shadow mode can facilitate (redirection-based) semantics manipulation and make it harder for introspection tools to handle. In particular, if the original instructions that would have been overwritten for detouring in direct mode are examined, they appear to be completely pristine and unchanged. In shadow mode, one basic way to achieve this is to tamper with function pointers (not code) in the related execution paths that contain those instructions and redirect their execution to our own code

Algorithm 1: TLB Poisoning

Input: Splitting Page Address (addr), Pagetable Entry for addr (pte)

1 `invalidate_instr_tlb` (*pte*);
2 pte = `the_shadow_code_page` (*addr*);
3 `mark_global` (*pte*);
4 `reload_instr_tlb` (*pte*);
5 pte = `the_orig_code_page` (*addr*) ;

to launch the attack. Although this scheme still requires the execution of our own attack code, it avoids the need to modify existing kernel code. In the following, we examine an advanced split-memory implementation that has been used in existing kernel rootkits, i.e., Shadow Walker [22].

Specifically, split-memory is achieved by intelligently poisoning the TLB cache for better stealthiness. This is made possible due to the presence of separate instruction cache (ITLB) and data cache (DTLB) on commodity *x86* processors. Both instruction fetch and data access are eventually achieved through ITLB and DTLB, respectively. The separation of ITLB and DTLB is intended to achieve better performance as instruction and data typically have different locality properties. In Algorithm 1, we show how this split-memory can be realized.

Basically, the algorithm invalidates the previous ITLB entry (this is achieved by the step 1) that points to the original code page and reloads it with the new shadow code page. The code in the shadow page implements the DKSM attack (by redirecting certain member fields of kernel data structures to some other locations). For the purpose of TLB poisoning, the shadow code page will not immediately contain a five-byte *jmp* (for the purpose of detouring the execution to the DKSM code as in the direct mode). Instead, it will contain another special *jmp* which returns back to the insertion point where the Algorithm 1 is executed. This special *jmp* is intended to facilitate the step 4 in two ways: Firstly, it causes this page to be loaded into the ITLB. Secondly, it returns the control back to the insertion point and resumes the execution to the redirection code for DKSM. Considering the entries in the TLB cache could be possibly invalidated by context switches, the algorithm marks the global bit to prevent this from happening (step 3).

In the meantime, due to the limited size of ITLB and DTLB caches, the cached entries, under TLB pressure, could be replaced due to the side-effect of another unrelated memory access. In order to have a reliable split-memory scheme, there is a need to re-populate the invalidated entries in the cache after they are replaced. Note a persistent TLB cache is possible if it is managed by software. However, if managed by hardware, this becomes much more challenging. The Shadow Walker rootkit [22] overcomes this challenge by leveraging certain protection bits associated with page table entries. In particular, by marking the original code pages not executable (*NX*), when they are to be executed, a page fault occurs, and a small piece of logic can execute to dictate the values that are supposed to be seen by whatever is accessing these pages. More specifically, if the pages are simply read or written to, these operations go through to the page containing the original values with no interception; however, for an instruction fetch, the ITLB will need to be reloaded to point to the memory page with the DKSM code. Therefore, one last important thing to handle is to re-gain the control to reload flushed TLB entries.

There are two main approaches to achieve that: (1) The first approach can directly modify the IDT (Interrupt Descriptor Table) to hijack the page fault handling routine. One limitation of this approach, however, is that the introspection tools can be enhanced to spot the modification of the IDT table. To mitigate that, we can choose to hijack a function pointer that is located in the execution path of page fault handling. The misuse of such a function pointer is significantly more stealthier than the direct modification of IDT (that has well-known values). (2) The second approach involves the debug registers (DRs). Specifically, one can use them to regain the control by placing breakpoints either on the page fault handling routine or those instructions that access the kernel data of interest. By doing so, we can minimize the changes to the system. The downsides, however, include the limited availability of DRs (which can greatly restrict the scale of launching the DKSM attack) and the possibility that the contents of these DRs can be inspected as well to detect such anomalies.

C. Return Scheme

Our next strategy is to apply the notion of return-oriented programming [7], [12], [20] that can effectively bypass existing code integrity protection schemes. As pointed out earlier, the *direct scheme* needs to modify existing kernel code and the modification will likely be trapped by code integrity protection schemes. Similarly, the *shadow scheme* requires the execution of attack code in the kernel space and the execution of unverified attack code, though common in compromised systems that are being introspected, will be prevented if the kernel code integrity is strictly enforced. To bypass these protection mechanisms, we naturally turn to return-oriented programming so that the proposed DKSM attacks become harder to defeat.

Note that in our prototype, we did not implement this return scheme. The reasons are that return-oriented programming can achieve Turing-completeness in performing malicious computation and existing research efforts have already successfully developed a return-oriented programming compiler [7], [12], [20]. Considering the main pur-

pose of this work is to expose the fundamental limitation of existing introspection techniques, we use a loadable kernel module to implement the other two modes. In this way, we can still model the same level of access a DKSM attack would have, if implemented based on return-oriented programming.

IV. Prototyping and Evaluation

We have implemented a proof-of-concept DKSM prototype and used it to attack a Ubuntu 9.04 system to demonstrate its capability to control the external view presented to introspection tools (e.g. XenAccess) as well as the internal view presented to various system management routines (e.g., top, ps, lsmod, and netstat). In the following, we present our prototyping details, three representative case studies, and related performance evaluation.

A. Prototyping Details & Case Studies

In our prototype, we first prepare all kernel data structure manipulation routines in a loadable kernel module (LKM). To launch the DKSM attack, we either directly modify the existing kernel code (the direct mode) or indirectly hijack the control flow (the shadow mode) and then invoke these routines in the LKM to manipulate kernel data structures. For simplicity, we focus our DKSM attack on redirection-based semantics manipulation in the context of direct mode.[2]

Before launching a DKSM attack, there are two questions which must be answered. (1) The first one is "what are the specific kernel data structures which should be chosen for the DKSM attack?" To answer this question, we select important kernel objects which are frequently attacked by existing kernel rootkits. Consequently, our goal here is to misrepresent the information about running processes, loaded kernel modules, and active network connections. (2) Once these kernel objects are determined, a follow-up question is "what are the related instructions that will access these data structures?" There are two complementary approaches. The first one is to analyze the kernel source code and identify those instructions that will access the kernel objects of interest. The second one is to profile the execution of OS kernel and locate every instruction that touches the chosen kernel objects. In our prototype, considering the convenience of system development and our past experience, we take the second approach. Specifically, we modified the open-source whole

[2]For the TLB-based shadow scheme, our prototyping experience indicates there exists a subtle architectural issue that affects the TLB flushing. Specifically, there exist a vast variety of virtual machine monitor implementations and running modes. Some of them will involve a world switch (from the guest to the virtual machine monitor and vice versa) in the process of re-gaining the control (Section III). A world switch for non-para-virtualized guests will flush the TLB, leading to unnecessary performance penalty. Fortunately, it is being avoided in recent processors with tagged-TLB support [4].

system emulator, QEMU $0.9.1$. By logging every memory access, the modified QEMU is used to show those instructions that access a given list of memory locations (containing the kernel objects of interest). We also point out that debug registers (DRs) can be used to profile these locations in a production system, avoiding the need of a modified QEMU.

After identifying the relevant instructions, we then launch the DKSM attack by loading a kernel module. When being loaded, the module initializes a shadow copy of the affected kernel objects and patches all these instructions in memory to redirect their accesses. With the redirection, the related memory accesses will go to the shadow copy instead of the original copy. In the following, we discuss three representative examples in detail on how a DKSM attack can be used to manipulate running processes, loaded modules, and active network connections, respectively.

1) Attack I: Process Manipulation: To manipulate running processes, we choose one important member field, i.e., PID, to demonstrate the DKSM attack. As described earlier, we first need to find out those memory addresses that contain the PID field and then profile the guest execution to locate all kernel instructions that access the PID field. At first, we thought this could be challenging due to the dynamic nature of running processes (as new ones will be dynamically created). Fortunately, it turns out that the kernel treats its access of the PID field in a generic way. Specifically, if we just profile the memory access for one particular PID, the identified instructions will be applicable for all PIDs in the system. This is not surprising since commodity OS kernels need to support dynamic kernel objects. As a result, there always exist a $1:N$ mapping from the kernel instructions to the accessed kernel objects. For our redirection purposes, there is a need to maintain a $1:1$ mapping between the original PID and the corresponding shadow copy. Accordingly, when shadowing the related PIDs, we will have to emulate the original kernel instruction, derive the exact PID that is being accessed, lookup the $1:1$ mapping, and then redirect the memory access.

When an external introspection tool applies the original data structure as a template to infer the guest state, it will be accessing the original memory locations. However, these locations are *no longer* used by the kernel. In other words, if we simply write an arbitrary number value (e.g., 42) into the original PID field, from that particular moment, introspection will report the PID as 42.

As a running example, the PID field of the *init* process (PID 1) on our Ubuntu 9.04 install is located at $0xdf8301ec$. This memory address is derived from the *task_struct* structure of the *init* process (located at $0xdf830000$) and the offset of the PID field in *task_struct* ($0x1ec$). With this address, we then exer-

```
root@inside:~# ps -A
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:00 watchdog/0
    5 ?        00:00:00 events/0
    6 ?        00:00:00 khelper
    7 ?        00:00:00 async/mgr
    8 ?        00:00:00 kintegrityd/0
    9 ?        00:00:00 kblockd/0
   10 ?        00:00:00 kseriod
   11 ?        00:00:00 khungtaskd
   12 ?        00:00:00 pdflush
   13 ?        00:00:00 pdflush
   14 ?        00:00:00 kswapd0
   15 ?        00:00:00 aio/0
   16 ?        00:00:00 crypto/0
   26 ?        00:00:00 kjournald
  111 ?        00:00:01 udevd
  960 tty4     00:00:00 getty
  961 tty5     00:00:00 getty
  967 tty2     00:00:00 getty
  970 tty3     00:00:00 getty
  973 tty6     00:00:00 getty
 1003 ?        00:00:00 syslogd
 1021 ?        00:00:00 dd
 1023 ?        00:00:00 klogd
 1041 ?        00:00:00 dbus-daemon
 1062 ?        00:00:00 sshd
 1114 ?        00:00:00 atd
 1139 ?        00:00:00 cron
 1158 ?        00:00:00 apache2
 1159 ?        00:00:00 apache2
 1160 ?        00:00:00 apache2
 1179 ?        00:00:00 apache2
 1234 tty1     00:00:00 getty
 1235 ttyS0    00:00:00 getty
root@inside:~#
```

(a) An inside view of running processes from *ps*

```
root@outside:/# process-list 5
[       1] **********
[       1] Direct
[       2] Kernel
[       3] Structure
[       5] Manipulation:
[       8]
[      13] Subverting
[      21] Virtual
[      34] Machine
[      55] Introspection
[      89]
[     144] for
[     233]
[     377] Fun
[     610] and
[     987] Profit
[    1597]
[    2584] by
[    4181]
[    6765] Sina
[   10946] Bahram,
[   17711] Xuxian
[   28657] Jiang,
[   46368] Zhi
[   75025] Wang,
[  121393] Mike
[  196418] Grace,
[  317811] Jinku
[  514229] Li
[  832040] and
[ 1346269] Dongyan
[ 2178309] Xu
[ 3524578] **********
[ 5702887]
[ 9227465]
[14930352]
root@outside:/#
```

(b) An external view of running processes from *XenAccess*

Fig. 2. A DKSM attack against running processes

cise possible code paths in the profiler (e.g., by running commands such as *ps*, *top*, *cat*, */proc/1/sched*, and so on) to identify related PID-accessing instructions. The profiling results indicate that there only exist two locations: $0xc01bf798$ (in kernel function *next_tgid*) and $0xc0117a52$ (in *proc_sched_show_task*). With the help of a disassembler, these two instructions are presented as follows:

```
0xc01bf798: 8b 80 ec 01 00 00  mov    0x1ec(%eax),%eax
0xc0117a52: 8b 81 ec 01 00 00  mov    0x1ec(%ecx),%eax
```

After identifying these two instructions, we can then load the LKM to create a shadow copy of the PIDs of all running processes and then replace these two with a *jmp* to our instrumentation code. Our instrumentation code essentially performs the memory access redirection from the original memory location to the corresponding shadow copy. During our prototyping, we experienced several caveats when these instructions were being patched. For example, the instruction that accesses the kernel object of interest may be less than five bytes in length, thus it is unable to accommodate the five-byte-long *jmp* instruction. As a result, subsequent instructions will need to be overwritten, which means our instrumentation code will have to compensate for the additionally overwritten instructions as a part of the instrumentation. After that, we need to seamlessly return the execution to where it is jumped from.

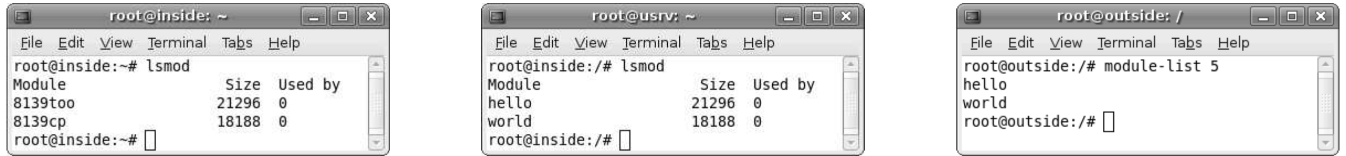
In our experiment, after loading the DKSM module to launch the attack, we run the command *ps -A* to get an *internal view* of running processes and use the *process-list* command that comes with XenAccess to obtain an *external view*. For comparison, the results are shown in Figures 2(a) and 2(b), respectively. We point out that the distorted information shown in Figure 2(b) "dramatizes" the fact that the external view of the VM can be arbitrarily changed and in practice, the attacker can easily create a more realistic yet deceptive view. In particular, the attacker can distort both internal and external views while maintaining their consistency (thus avoiding the view comparison-based detection [13]). An astute reader may also observe that the PIDs in Figure 2(b) actually follow the Fibonacci sequence, showing the fact that there is no synthetic limitation (other than the actual 32-bit width restriction) when launching the DKSM attack against the 32-bit PID field of the task. Our experience confirms that similar steps can be used to handle the redirection of almost all other data types.

2) Attack II: Manipulating Loaded Modules: In this experiment, we apply the DKSM attack to manipulate the views about loaded kernel modules. Specifically, we choose the module name field to demonstrate our attack.

Redirecting a name string is similar to, but slightly different from, the redirection of non-string types. In particular, most of our previous steps remain the same. We first find out the memory address that contains the module names. Using a kernel module *8139cp* as an example, its name field on our system is located at $0xe084d5cc$. Similarly, this memory address is derived from the module structure for the *8139cp* module (located at $0xe084d5c0$) combined with the offset of the name field in the module structure ($0xc$). Just as we did previously, we exercised possible code paths in the profiler (e.g., by running commands such as *lsmod* and *cat /proc/modules*) to identify related module-name-accessing instructions. The profiling results indicate that there only exists one instruction at $0xc0248131$ (in the *string* routine):

```
0xc0248131: 89 e5     mov    %esp,%ebp
```

We can see that the *string* function is responsible for the module name access. However, if we overwrite this function, we would intercept the checking of every single string that the kernel deals with. This is certainly not a tenable and efficient solution. Instead, what needs to occur is the movement of our redirection to one semantic level higher and for us to identify the corresponding call sites that invoke the *string* routine. As a result, we identify



(a) An inside view of loaded modules from the first run of *lsmod* before launching the attack

(b) An inside view of loaded modules from the second run of *lsmod* after launching the attack

(c) An external view of loaded modules from *XenAccess* after launching the attack

Fig. 3. A DKSM attack against loaded modules

the *m_show* routine. The related instructions are located at $0xc013c542$ (shown below). By overriding the value of *eax*, we can successfully control how the module names are eventually printed.

```
0xc013c542: 83 c0 08    add    $0x8,%eax
0xc013c545: 89 44 24 08 mov    %eax,0x8(%esp)
```

If we look back at the previous attack on running processes, there are two instructions (at $0xc0249959$ and $0xc02481bc$) that access the process name field. They appear in frequently used routines, *strnlen* and *string*. Interestingly, by moving our redirection one level higher, we identify a common instruction (*add $0x2d4,%edx*) at $0xc0185651$ in the *get_task_comm* function. In our instrumentation, we simply examine the current content in *edx*, use it to locate the corresponding shadow string, and reload it with the shadow copy. By doing so, we can not only handle both instructions, but also avoid redirecting an instruction that occurs in a very frequently executed loop.

```
0xc0249959: 80 38 00    cmpb   $0x0,(%eax)
0xc02481bc: 0f b6 04 13 movzbl (%ebx,%edx,1),%eax
```

To demonstrate our attack, we first run the command *lsmod* to get an authentic *internal* view of loaded modules. Then we load the DKSM module, re-run the same command to get a manipulated *internal* view. After that, we also use the *module-list* command that comes with XenAccess to obtain an *external* view. The results are shown in Figures 3(a), 3(b), and 3(c), respectively. From the figure, our attack module successfully controls both internal and external views. It is important to note that a number of introspection-based tools [11], [13] have been developed to compare its external view with an internal view and any discrepancy will indicate the presence of a hidden malware. The coordinated control of both internal and external views is needed by DKSM to foil such a cross-view comparison.

3) Attack III: Manipulating Network Connections: Next, we present our DKSM attack against network connections. Note that the manipulation of network connection information is remarkably similar to what we have discussed so far. Particularly, the redirection of port numbers is almost the same as the redirection of the process' PID in Section IV-A1. The manipulation of the name of a running process which owns a network connection is extremely similar to the process/module name redirection in Section IV-A2. This similarity highlights an intriguing aspect of any DKSM attack: For different kernel objects that are being affected by the attack, though the underlying semantics might be vastly different, they share a limited set of common attack mechanisms. In the following, we only present an abbreviated description of this experiment.

In our experiment, we choose to hide an active network connection as the demonstration. We found that the *tcp4_seq_show* function is the one that iterates across the list of network connections. Therefore, a simple attack on the iteration code is designed to achieve the intended results. After the attack, we ran the *netstat* command to list the TCP connections in the LISTEN state and wrote a *network-list* utility based on XenAccess to list the TCP ports as well as the applications that own these ports. The results are shown in Figures 4(a) and 4(b), respectively. As we can see, one active TCP port, i.e., 80, is hidden from the external view.

B. Performance

	DKSM (NO)	DKSM (YES)	Overhead
Apache (#reqs/sec)	911.341	886.483	2.7%
Kernel Compilation (seconds)	247.788	248.777	0.4% (User)
	189.835	194.556	2.5% (System)
	449.93	452.35	0.5% (Total)

TABLE I. Summary of experiments

To evaluate the performance impact from the proposed DKSM attack, we use a default Ubuntu 9.04 32-bit install on a standard Dell Optiplex 760 desktop machine. The desktop has 4GB of memory and an Intel Core2 Quad CPU Q9550 processor, running at 2.83GHz. We choose two performance measurement tasks: the Apache HTTP throughput benchmarking tool and kernel compilation. Our HTTP benchmarks were performed with a one minute duration, at a concurrency level of 4. The kernel compilation runs consist of compiling the kernel ten times and taking the average of the reported time. Each measurement task is performed twice: one with DKSM disabled and another enabled. When enabled, the DKSM attack achieves the manipulation of running processes, loaded modules, and active network connections (Section IV-A). We summarize our results in Table I.

From the table, we can see the DKSM-infected system causes a 2.7% slowdown in the measured HTTP throughput. In our experiments, we noticed up to a 5% variation

```
root@inside:~# netstat -ano | grep LISTEN | grep tcp
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      off (0.00/0/0)
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      off (0.00/0/0)
root@inside:~#
```

(a) An inside view of network connections from *netstat*

```
root@outside:/# network-list 5
       Network List

Port  Process
   22 sshd

* 1 ports detected
root@outside:/#
```

(b) An external view of network connections from *XenAccess*

Fig. 4. A DKSM attack against network connections

simply across multiple sets of runs. For the kernel compilation, the overhead of the total elapsed time to compile the kernel is 0.5%. Again, a higher variation was noticed across multiple compilation runs, leading us to firmly believe that this overhead is statistically insignificant.

V. Discussion

In this section, we re-visit the nature of the proposed DKSM attack and aim to better understand the limitations of existing introspection tools. This analysis is necessary as it can lead to countermeasures that can be potentially deployed to defend against DKSM and insights for the development of next-generation, reliable introspection tools.

In the various instantiations of DKSM (Section II), we can see that the success of DKSM is directly proportional to its scope and capability of kernel data access. This ratio directly translates into the efficacy of the attack. For example, if DKSM was unable to redirect or manipulate a particular field or data structure, then it would be much harder to attack such kernel data, and consequently it might be unable to foil various types of introspection analysis. This results in two potential limitations of DKSM which can be leveraged for defense purposes.

Unmanipulatable structures The first one involves the inability of DKSM to redirect certain structures that are specified and used by the CPU directly. For example, the global descriptor table (GDT), the interrupt descriptor table (IDT), the task state segment (TSS), and so forth. These structures, once loaded into the CPU cannot be changed implicitly. To change them, an explicit reload operation is necessary. For example, the *lidt* instruction will reload the IDTR to a given memory address. Fortunately, these malicious reloads can be easily detected and defeated. Similarly, for an introspection tool, it is important to start from the unmanipulatable data structure as a base and then gradually expand from it to reliably infer other guest states.

Untamperable control flow To influence the kernel's interpretation of a particular kernel object, DKSM needs to maintain its own *actual* view on how the object should be accessed. To do that, there is a need for DKSM to hijack the control flow, either by directly modifying existing kernel code (the direct mode) or indirectly tampering with a function pointer (the shadow mode) or a return address (the return mode). As such, if a full kernel control-flow integrity (CFI) guarantee can be made about a system, such a guarantee will disallow DKSM to execute in the first place. Unfortunately, there are no working systems [17] developed yet to guarantee the kernel CFI, due to the fact that the enforcement of kernel CFI is much more challenging than the user-level counterparts [3] (e.g., because of the support of multi-tasking and asynchronous interrupts in commodity OS kernel design). Alternatively, a weaker form of semantic integrity [5] can be used to detect the violation of kernel data invariants. Recent efforts [6] have made encouraging progress toward this direction by inferring these invariants.

More fundamentally, if we re-examine the nature of introspection, an external introspection tool aims to analyze a guest which is not trusted. However, it still depends on the guest-maintained memory state and expects the untrusted guest to respect the kernel data structure templates, therefore leading to a *trust inversion* problem. This problem fundamentally explains the effectiveness of our attack and equivalently the fragility of existing introspection solutions. For the very same reason, we also believe that existing memory snapshot-based memory analysis tools and forensics systems [1], [8], [10] share the same limitation.

From another perspective, in this paper, we have so far only explored the spatial aspect of DKSM (i.e., the layout of a data structure), not the temporal aspect. Considering the dynamics of a guest OS, an introspection-based analysis of a running guest typically requires a period of time to complete and is thus temporally limited in its capability to obtain a consistent view. To partially address that, some introspection tools such as VIX [16] choose to pause guest execution while performing introspection activities. However, this adversely perturbs the execution of the guest VM. Further, the asynchronous and independent nature of external introspection still implies it may not be mutually excluded when the guest is running in a critical section, resulting in an inconsistent view. It is part of our future work to assess the extent and scale of this limitation.

VI. Related Work

VM Introspection can be executed in two different ways. The first one involves the introspection completely running outside of the guest. Several examples of the external approach exist [2], [5], [11], [13], [14], [16], [23]. This approach benefits from a much stronger level of isolation, and thus protection. Unfortunately, because this introspection is performed outside of the guest, its view is also an external one and cannot benefit from the

implicit advantages afforded to the internal view. Namely, an external view has to bridge a significant semantic gap [9]. This results in a much more complex implementation of introspection techniques with its VMM-level view of the guest. Several systems such as Livewire [11], VMwatcher [13], and VMwall [23] have to reconstruct the semantics of what is executing within the guest and thus are vulnerable to the proposed DKSM attack.

The second approach to VM introspection is to take a hybrid approach by having two entities, one inside and another outside. The goal here is to obtain the advantages of having a semantic-rich view of the guest with the help of an internal entity while still protecting the internal entity from being corrupted. SIM [21] is a recent research system that is moving towards this direction. However, in its current implementation, the current internal agent, though running inside the guest context, still suffers from the semantic gap as it is designed not to rely on any existing kernel code. As such, it is still vulnerable to the proposed DKSM attack. From another perspective, as it is running inside the guest, it has unique advantages that can be potentially leveraged to mitigate the DKSM attack. We plan to explore this possibility in our future work.

VII. Conclusion

In this paper, we have shown that current VM introspection techniques are subject to an attack called DKSM. By violating their basic assumption about the use of underlying kernel data structures, a DKSM attack can change the syntax and semantics of kernel data structures in a running guest. We have developed a proof-of-concept prototype and used it to manipulate important system information (e.g., running processes, loaded kernel modules, and active network connections) to successfully foil existing introspection tools into reporting false information. By exposing this fundamental limitation, we aim to examine the challenges as well as opportunities for the development of next-generation, reliable VM introspection techniques.

Acknowledgments The authors would like to thank the reviewers for their insightful comments. This work was supported in part by the US AFRL grant FA8750-09-1-0224 and the US NSF grants 0852131, 0855141, 0855297, and 0952640. Any opinions and findings expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL and the NSF.

References

[1] Volatile systems. https://www.volatilesystems.com.

[2] Xenaccess library. http://code.google.com/p/xenaccess/.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):1–40, 2009.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 3.14 edition, September 2007.

[5] F. Baiardi, D. Cilea, D. Sgandurra, and F. Ceccarelli. Measuring semantic integrity for remote attestation. In *Proc. of the 2nd Conference on Trusted Computing*, 2009.

[6] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proc. of the 2008 ACSAC*, pages 77–86, Washington, DC, 2008.

[7] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proc. of the 15th ACM CCS*, 2008.

[8] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proc. of the 16th ACM CCS*, 2009.

[9] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proc. of the 8th HotOS Workshop*, 2001.

[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust Signatures for Kernel Data Structures. In *Proc. of the 16th ACM CCS*, pages 566–577, 2009.

[11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of the 2003 NDSS*, February 2003.

[12] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypasssing Kernel Code Integrity Protection Mechanisms. In *Proc. of the 18th USENIX Security Symposium*, 2009.

[13] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through VMM-based "Out-of-the-Box" Semantic View Reconstruction. In *Proc. of the 14th ACM CCS*, 2007.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proc. of the 2006 USENIX Annual Technical Conference*, Berkeley, CA, 2006.

[15] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proc. of the 17th conference on Security symposium*, 2008.

[16] K. Nance, M. Bishop, and B. Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6(5):32–37, 2008.

[17] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. of the 14th ACM CCS*, 2007.

[18] R. Riley, X. Jiang, and D. Xu. An Architectural Approach to Preventing Code Injection Attacks. In *Proc. of the 37th DSN*, pages 30–40, 2007.

[19] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proc. of the 11th RAID*, 2008.

[20] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of the 14th ACM CCS*. ACM, 2007.

[21] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM Monitoring using Hardware Virtualization. In *Proc. of the 16th ACM CCS*, 2009.

[22] S. Sparks and J. Butler. Shadow Walker: Raising the Bar For Windows Rootkit Detection. *Phrack*, 11(63), 2005.

[23] A. Srivastava and J. Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proc. of the 11th RAID*, Berlin, Heidelberg, 2008.

[24] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance. *IEEE Trans. Dependable Secur. Comput.*, 2(2):82–92, 2005.

[25] VMware. VMware VMsafe Security Technology. http://www.vmware.com/technical-resources/security/vmsafe.html.