CMPS223 Final Project
Virtual Machine Introspection Techniques

Michael Sevilla
University of California, Santa Cruz
msevilla@soe.ucsc.edu

December 4, 2012

1 Introduction

This work is a survey of Virtual Machine (VM) introspection, a necessary tool when utilizing VMs for security purposes. In the the rest of this section, we discuss traditional techniques for dealing with malware and the appeals of using a VM in a security context. In Section 2, we outline the main problem for using VMs for security called the semantic gap. In Section 3, we analyze 3 related approaches to VM introspection and focus on their advantages and limitations. In Section 4 we show how to leverage existing tools to implement a small rootkit detector.

1.1 Traditional Intrusion Detection and Malware Analysis Systems

Intrusion detection systems (IDSs) monitor the observable properties of a system (state, transitions, I/O) [5] to determine if a system is under attack. [5] classifies the effectiveness of IDSs along two axis' - visibility and resilience to attack. A highly visible system easily exposes system state and events - this makes it difficult for malicious software, known as malware, to evade detection. A highly resilient system provides good isolation between the IDS and the attacker so the IDS is less likely to be under attack or compromised.

The dominant IDS architectures, host-based intrusion detection systems (HIDS) and network-based intrusion detection systems (NIDS), have severe flaws, as shown in Figure 1. HIDSs operate at either user or kernel space, giving them good visibility because they can report process lists, network connections, files, etc. Unfortunately, HIDSs do not have high resilience because their control and the data structures they rely on can be taken over by an attacker - a compromised IDS is useless. NIDS monitor traffic through the system so they have good resilience to attack since the IDS is isolated

	Visibility	Resilience
HIDS	✓	✗
NIDS	✗	✓
VM IDS	✓	✓

Figure 1: *Comparing the advantages of using a VM IDS over the traditional IDSs. VMs offer large advantages for IDSs because of their visibility into the guest OS state and their resilience to attack.*

from the untrusted host. Unfortunately, NIDs have poor visibility into the actual host system - they can examine incoming and outgoing packets but they cannot see how they are being used in the system [6].

Malware analysis systems attempt to extract information from the system while malware runs. The goal is to characterize the behavior of the malware in the hopes of detecting and destroying it in the future. Not surprisingly, malware analysis systems have the same requirements as an IDS - they must be resilient to attack and visible to an outside entity [2].

1.2 The Argument for Virtual Machines

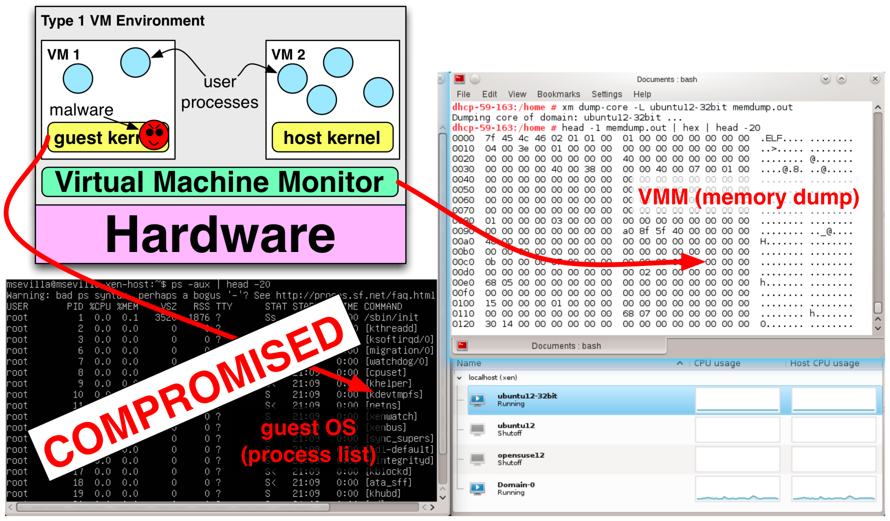
Virtual machines have become active platforms for intrusion detection and malware analysis because they offer both visibility and resilience, as noted in Figure 1.

Virtual machine systems consist of three parts, as shown in the upper left corner of Figure 2. The first part, the VM, is a duplicate of a physical machine. The second part is the guest operating system (OS), which is simply an OS running inside of a VM. Finally, the virtual machine monitor (VMM) is a thin layer of software that manages VMs. The goal of a VMM is to multiplex physical resources for its VMs and it runs directly on the hardware. In general, VMs and VMMs strive to be efficient, simple, general, and isolated [6, 5, 9]. Up until 2003, VMs had been used for server consolidation, testing/developing OSs, and hardware virtualization [6].

In the context of security, using a VM is appealing because it provides a high degree of a resilience with a moderate amount of visibility. The idea is to move the IDS away from the malware and into the VMM or the host OS. According to [5], a well designed VM system will provide:

- isolation (i.e. between VMM and VMs)
- inspection (i.e. access to hardware state[1])
- interposition (i.e. privileged functionality)

[1]Hardware state includes, but is not limited to, CPU state (registers), memory, and I/O devices (storage, I/O registers).



```
dhcp-59-163:/home # xm dump-core -L ubuntu12-32bit memdump.out
Dumping core of domain: ubuntu12-32bit ...
dhcp-59-163:/home # head -1 memdump.out | hex | head -20
0000  7f 45 4c 46 02 01 01 00  01 00 00 00 00 00 00 00  .ELF.... ........
0010  04 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  ..>..... ........
0020  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........ @.......
0030  00 00 00 00 40 00 38 00  00 00 40 00 07 00 01 00  ....@.8. ..@.....
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0050  00 00 00 00 00 00 00 00  00 [illegible]
0060  00 00 00 00 00 00 00 00  00 [illegible]
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0080  01 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0090  00 00 00 00 00 00 00 00  a0 8f 5f 40 00 00 00 00  ........ .._@....
00a0  48 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  H....... ........
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
00c0  0b 00 00 00 [illegible]                 00 00 00  ........ ........
00d0  00 00 00 00 00 00 00 00  00 02 00 00 00 00 00 00  ........ ........
00e0  68 05 00 00 00 00 00 00  00 00 00 00 00 00 00 00  h....... ........
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0100  15 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0110  00 00 00 00 00 00 00 00  68 07 00 00 00 00 00 00  ........ h.......
0120  30 14 00 00 00 00 00 00  00 00 00 00 00 00 00 00  0....... ........
```

localhost (xen)
- **ubuntu12-32bit** Running
- **ubuntu12** Shutoff
- **opensuse12** Shutoff
- **Domain-0** Running

```
msevilla@msevill[illegible]xen-host:~$ ps -aux | head -20
Warning: bad ps synt[illegible] perhaps a bogus '-'? See http://pro[illegible]s.sf.net/faq.html
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT ST[illegible]  [illegible]ME COMMAND
root         1  0.0  0.1   352[illegible] 1876 ?        Ss   [illegible]  [illegible]00 /sbin/init
root         2  0.0  0.0      0     [illegible] [kthreadd]
root         3  0.0  0.0      0     0 ?        [illegible] [ksoftirqd/0]
root         6  0.0  0.0      0     [illegible]  [illegible]00 [migration/0]
root         7  0.0  0.0      0     [illegible]        0:00 [watchdog/0]
root         8  0.0  0.0  [illegible]  21:09   0:00 [cpuset]
root         9  0.0  0.[illegible]   S<   21:09   0:[illegible] [khelper]
root        10  0.[illegible]        S    21:09   0:00 [kdevtmpfs]
root        11 [illegible]           S<   21:09   0:00 [netns]
root    [illegible]     0 ?        S    21:09   0:00 [xenwatch]
root    [illegible]     0 ?        [illegible] [xenbus]
root    [illegible]   0     0 ?        [illegible] [sync_supers]
root    [illegible].0     0     0 ?        [illegible] [bdi-default]
root    [illegible].0  0.0    0     0 ?        [illegible] [integrityd]
root        17  0.0  0.0      0     0 ?        S<   21:09   0:00 [kblockd]
root        18  0.0  0.0      0     0 ?        S<   21:09   0:00 [ata_sff]
root        19  0.0  0.0      0     0 ?        S    21:09   0:00 [khubd]
```

Figure 2: *On the left (black screen), we have the guest OS reporting its processes. On the right (white screens), we have the VMM monitor showing the physical memory and CPU usage. Malware, pictured as the red process, can compromise an OS if it penetrates the kernel. A compromised OS can report incorrect state, so we must use the hardware information provided by the VMM to reconstruct high level state.*

If the system provides these properties, it is easy to offer resilience since the host is isolated from the attacker via the protection domains. Although the visibility is better than a NIDS, it is still NOT good enough to properly implement effective IDSs and malware analysis systems. Work attempting to improve the visibility in a VM system has produced a process called introspection, which, although not perfect, can help improve the visibility of the system.

2 Virtual Machine Introspection

To extrapolate the behavior of both the malware and the infected system, a programmer, operating in the context of the host OS, needs to extract state and event information from the guest OS. In other words, the programmer needs to inspect the VM from the "outside" to analyze the software running on the "inside" [5]. Unfortunately, a well known problem called the semantic gap makes this process difficult.

2.1 Problem: The Semantic Gap

In VM systems, the semantic gap is the information mismatch between the guest OS's high level semantic state and its low level bits [4]. From outside the guest OS, it is difficult to correlate the guest OS's software activity with its hardware events.

In the correct context, these problems are not difficult. From inside the guest OS, it is trivial to gather information about the guest OS software state by using the guest OS programs and APIs. Similarly, from outside the guest OS, it is simple to efficiently gather information about the hardware state from the VMM [9, 5].

Unfortunately, when VMs are used for security, the guest OS cannot be used to bridge the semantic gap because it may be compromised. Malware that has penetrated the guest OS can alter how it exposes its state, as shown in Figure 2. Therefore, we cannot trust the guest OS programs and APIs so we must get the same functionality outside the VM using the bits representing memory, processor activity, and register values exposed by the VMM [2].

2.2 What is introspection & how will it help us?

Introspection is the process of extracting the guest OS state from outside the VMM [5]. An introspection tool attempts to bridge the gap between the hardware and software layers. Ultimately, this will enable us to analyze malware behavior so that we can reason about ways to detect, prevent, and fix corrupt systems.

Introspection has many applications, aside from IDSs and malware analysis. In [9, 2], the authors list applications and research projects with a need for efficient introspection, including trusted computing, policy enforcement, Nettops, virtual distributed monitoring, logging and replay, archiving attacks, forensic analysis, integrity checking, honeypot analysis and VMM sensors.

2.3 Why introspection is difficult

Traditional introspection[2] is difficult because most implementations require:

- extensive knowledge, which reduces generality
- manual inspection, which increases complexity
- a high degree of flexibility, which is costly

Traditional introspection requires extensive knowledge of the guest OS data structures and behavior. A common technique is to examine physical memory; an introspection tool can "walk" memory and leverage knowledge of the OS algorithms and data structures to recreate the high level semantic information. Knowledge of the OS can include layout, location of kernel

[2]This is a term I use to describe the early VM IDS architectures. These usually date to around 2003, since this is when [5, 6], the pioneering work in VM IDS systems, were published.

structures, kernel memory layout, and global variable layout. As a result, generality is poor since the introspection tool is tied to the guest OS [2]. This problem is exacerbated if the guest OS is not open-source or if the developers wish to support additional types of VMs and VMMs [3].

Designing a tool that performs manual inspection is especially complex, leading to performance and correctness challenges. Comparing system configurations, locating and using the OS kernel structures, and managing the data slows down the system and makes it difficult to launch these techniques in an “online” fashion. The complexity also increases the likelihood of buggy introspection behavior since designing manual inspection introspection is tedious and slow. To make things worse, the collection method (i.e. when to snapshot memory) and its frequency can increase the amount of information that needs to be managed and exposed [2].

Finally, flexibility of the tool functionality is desirable - most programmers would like the widest range of reportable information. Unfortunately, the higher the functionality, the higher the cost.

3 Approaches & Achievements

The evolution of introspection has had a predictable trajectory. It started with analyzing crash dumps and utilities, which requires debugging with symbols and kernel data structure knowledge [4]. Then modifying the kernel to monitor simple OS constructs, such as system calls [6], became prevalent. As VMMs, like Xen [9] and VMSafe [3] became more sophisticated, they started to offer APIs to easily expose hardware. Today’s introspection tools try to automatically and efficiently bridge the gap between high level software and low level hardware events.

What follows is an examination of 3 recent papers that aim to further reduce introspection obstacles by collecting and interpreting information automatically.

3.1 Leveraging Digital Forensics

Dolan-Gavitt at al. [3] propose using digital forensics to help solve current VM introspection problems.

Forensic memory analysis (FMA) aims to answer many of the same questions that plague introspection tools - what happened in the system, what state is the machine in, what transition it is making, etc. To aid the digital forensic programmer, tools have been developed to parse, reconstruct, and re-walk the hidden OS structures in physical memory dumps. [3] notices that FMA tools have already done much of the work to bridge the semantic gap and they conclude that it is foolish to redesign manual inspection tools for introspection since the process is error prone, time-consuming, and tedious.

FMA tools cannot be used as is because VM security systems need to perform "live analysis" to implement monitoring techniques that can analyze and detect events as they occur. FMA occurs after a security incident is suspected to have occurred, so there is no notion of snapshotting or incremental updates. If systems snapshot or store data incorrectly, the introspection tool may induce data explosion and poor performance, as was shown in [10].

To use FMA tools for live security analysis, [3] hooked up a FUSE filesystem to the Xen VMM - the memory is dumped to a file and then introspection tools are used to glean important data structures. To show this, they re-implemented VMWall (filters traffic based on application) by utilizing existing capabilities in an open-source FMA framework. The implementation was declared a success, as it detected system call table and IRP hooking.

Discussion

The authors provide a convincing argument for the effectiveness and practicality of using FMA tools in VM introspection. Using FMA tools prevents developers from wasting time coding redundant functionality and this model, of continually adding to what others have done is a common motif in computer science.

The simplicity of the approach allows other facets of computer science to contribute to introspection. For example, deduplication techniques could be applied to memory snapshots so that only differences are saved instead of the entire memory dump. Other computer science fields, like data mining, machine learning, and optimal control, have opportunities for efficiently training these systems instead of requiring knowledge of the OS internal workings.

3.2 Virtuoso

Virtuoso [2] is a big extension of [3] and tries to bridge the semantic gap automatically. The authors lament the fact that VM-based intrusion detection, forensic memory analysis, and low artifact analysis all require a detailed knowledge of the guest OS in order to figure out the current state of the guest OS in the VM. They lay out similar arguments to section 3.1 and append the argument that since we are stuck with this process, we should at least make it easier to use.

Virtuoso automatically generates host OS programs that extract information from the guest OS. The tool dynamically traces the execution of trusted guest OS programs (*training programs*), like Linux's `ps`, and produces python code and data. This code and data is executed by the host OS to get information from an untrusted guest OS.

To do this, the authors inserted logging functions into QEMU to spit out *instruction traces* about the physical memory address and page table entries for load and store functions. These instruction traces are analyzed by a *trace analyzer*, which produces the data and tools.

They tested their solution on the basis of generality, reliability, and resilience to attacks aimed at output programs. For generality, they generated 6 training programs for three different OSs and verified that the generated programs were correct by hand. For reliability, they test drove on Windows while tracing `pslist` and then checked to see if the output was the same as the training program for that image. For resilience, they used a hacker toolkit to hide a process in the OS and determined if it could be detected.

The authors concluded that Virtuoso exhibited generality, reliability, resilience, and unsatisfactory performance (not fast enough for "live analysis").

Discussion

Virtuoso transforms a tedious and time-consuming task into a task for an average programmer but the system raises more questions than answers.

- limitations: many, like [1], despite system complexity - good tradeoff?
- scope: trace tools restricted to system call data - too narrow?
- performance: is slow and ignores training - how is this feasible?
- dependencies: partitions at kernel/user space - aren't these related?

Even the assumption that an average programmer[3] could use this system is doubtful. To navigate Virtuoso, the programmer needs knowledge of the Virtuoso components and the involved VMM, VMs, and OSs. Furthermore, the programmer has to know what he/she wants. A big problem with dealing with malware is that it is stealthy and unpredictable - how can we expect the programmer to know, *a priori*, which tool to use?

3.3 VM Space Traveler

VM Space Traveler (VMST) [4] builds on Virtuoso's promising framework by increasing automation and transparency. The tool automatically bridges the semantic gap to generate VMI tools - there is no need to specify which tools the user wants.

While Virtuoso requires a little intervention and tool framework knowledge, VMST automatically generates secure introspection tools with path coverage and higher data capacities. The methodology is far different than Virtuoso because it uses system wide instruction monitoring to redirect important kernel data. The steps are: (1) identify system call execution context, (2) identify redirectable data, and (3) redirect kernel data

[3]It is entirely possible that I am not an average programmer.

Identifying system call execution contexts involves identifying the system call and relevant kernel data. Identifiyng the redirectable data requires figuring out the kernel data that is useful for introspection. Redirecting the relevant kernel data is the final step. This tool must be run in parallel with the in-product VM.

The authors concluded that VMST provides functionality, performance, and generality. Experiments show that the tool incurs a 9.3X overhead in the worst case for regular use.

Discussion

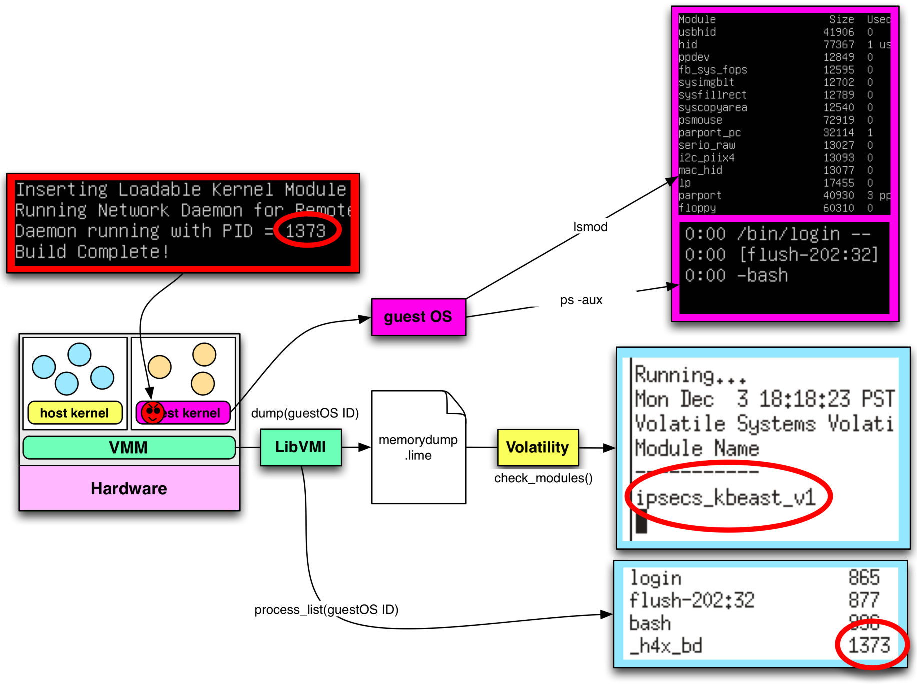
VMST does well to automatically bridge the gap and uses sophisticated user and kernel space techniques to reuse the OS. The idea of redirecting the desired introspection functionality seems like a solid option, especially for liveness and correctness proofs.

Although VMST provides small improvements over Virtuoso it's complexity seems to outweigh the advantages. First, it does not seem as transparent as they declare, and the complexity makes extensibility of the code more difficult. For example, extending this to different system-call interfaces and semantics seems daunting. Second, there is little discussion for storing the data - what data structures need to be maintained and managed to support this functionality? The system already seems to have poor performance - wouldn't storing data make this worse? Finally, creating an excessive amount of tools seems time consuming and superfluous - it may not be the case that the administrator wants all these tools - there should be a selectable subset to improve performance and storage overheads.

4 An Implementation: A Return to Leveraging Introspection & FMA Tools

Introspection tools are trending towards complexity; in this section we attempt to show how current tools and simple approaches can produce excellent results without extensive knowledge of the system or the tools. In effect, we are trying to shift the focus back to simplicity and leveraging existing tools, like in Section 3.1.

We used LibVMI [8], Volatility [11], and KBeast [7] to reveal a hidden process and the accompanying module. LibVMI is a "live analysis" introspection tool that uses a VMM to read/write memory. We run LibVMI on Xen, with Ubuntu 12.04 running Linux 3.2 as the guest OS and OpenSuse 12 running Linux 3.4 as the host OS. Volatility is an FMA toolkit that extracts kernel data structures from physical memory (RAM) dumps. KBeast is a stealth rootkit, loaded as a kernel module, which launches a hidden backdoor for remote access. Using system call hooks, KBeast hides files,



```
Inserting Loadable Kernel Module
Running Network Daemon for Remote
Daemon running with PID = 1373
Build Complete!
```

```
Module                  Size  Usec
usbhid                 41906  0
hid                    77367  1 us
ppdev                  12849  0
fb_sys_fops            12595  0
sysimgblt              12702  0
sysfillrect            12789  0
syscopyarea            12540  0
psmouse                72919  0
parport_pc             32114  1
serio_raw              13027  0
i2c_piix4              13093  0
mac_hid                13077  0
lp                     17455  0
parport                40930  3 pp
floppy                 60310  0
```

```
0:00 /bin/login --
0:00 [flush-202:32]
0:00 -bash
```

```
Running...
Mon Dec  3 18:18:23 PST
Volatile Systems Volati
Module Name
-----------
ipsecs_kbeast_v1
```

```
login             865
flush-202:32      877
bash              896
_h4x_bd          1373
```

Figure 3: *Lists from actual screenshots of a system running our program* `monitor-VM.sh`*. The KBeast rootkit (red process), hides a process (pid 1373) and module from the guest OS.* `monitor-VM.sh` *combines LibVMI and Volatility and uses their APIs to expose the intruder.*

processes, sockets, connections, and more.

We wrote a small program, called `monitor-VM.sh`, to detect and alert the user to a hidden module, as shown in Figure 3. The shell script uses LibVMI to dump memory to a file. Then it runs the `check_list_modules()` Volatility function over the dumped memory. `check_list_modules()` uses `sysfs`, which exports `/sys/module` information, to determine which modules were removed from the module list but still active [12].

Using this technique, we were also able to identify the hidden process and module with less than 10 lines of code, as shown in Figure 4.

4.1 Lessons Learned

Although their presence in the community (forums and blogs) is strong, these tools are still works-in-progress and setting them up is difficult and time consuming. Xen favors specific distributions (openSUSE), LibVMI requires finding kernel `struct` offsets, and Volatility requires `ziping` together symbol tables and build libraries. To make matters worse, the tools, especially LibVMI, are not user-friendly and they outputs poor error messages,

```
#! /bin/sh
while [ 1 ]
do
    ~/vmitools/dump-memory ubuntu12-32bit memdump.lime
    echo "Running..."
    date
    ~/vol.py -f memdump.lime --profile=ubuntu linux_check_modules
    rm memdump.lime
done
```

Figure 4: *The source for* `monitor-vm.sh` *is only 9 lines long. The script continually dumps memory with LibVMI and checks for hidden modules with Volatility.*

especially when checking arguments.

Used separately, the limitations of these tools are easily exposed. LibVMI has tools that examine process lists and loaded modules but they traverse the data structures in the same fashion as the OS. This will produce results that are the same as the guest OS since the faulty structures are being used. As a result, in initial tests with LibVMI and KBeast, we could see the hidden process but not the loaded module. Volatility is not designed for "live analysis" so dumping memory with its corresponding tool, LiME, is very slow and requires loading a loadable kernel module. The good news, as we showed in this section, is that these tools used in conjunction are extremely powerful.

Conclusion

In this work, we did an in-depth analysis of current introspection techniques and their tradeoffs. Malware can force a guest OS to report incorrect information so introspection tools were created to help bridge the semantic gap between the high level semantic state in software and the low level bits in hardware. Introspection techniques have trended towards using FMA and automatically creating introspection programs, as shown by Virtuoso and VMST. We noticed that the complexity of these systems has increased yet the use of the systems still required advanced knowledge of the components and the approach. As a result, we implemented a small program that combined an underdeveloped introspection tool (LibVMI) with a sophisticated FMA tool (Volatility). Our tool, called `monitor-VM.sh`, successfully detected hidden process and modules in a guest OS.

References

[1] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 82–91, Washington, DC, USA, 2010. IEEE Computer Society.

[2] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[3] B. Dolan-Gavitt, B. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[4] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 586–600, Washington, DC, USA, 2012. IEEE Computer Society.

[5] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[6] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *Proceedings of the 30th EUROMICRO Conference*, EUROMICRO '04, pages 520–525, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Packetstorm. http://packetstormsecurity.org/files/108286/kbeast-kernel-beast-linux-rootkit-2012.html.

[8] B. Payne. http://code.google.com/p/vmitools/.

[9] B. D. Payne and W. Lee. Secure and flexible monitoring of virtual machines. In *ACSAC*, pages 385–397, 2007.

[10] J. Toldinas, D. Rudzika, V. Štuikys, and G. Ziberkas. Rootkit detection experiment within a virtual environment. *Electronics and Electrical Engineering–Kaunas: Technologija*, NA(8):104, 2009.

[11] Volatility. http://code.google.com/p/volatility/.

[12] Volatility, September 2012.