Orange Labs Caen

Final Internship Report

A Deep State-of-The-Art about Mainstream VMI and theirs Manipulation

Author:
Qipeng Song

Supervisor:
Sylvie Laniepce

A technical report submitted in fulfilment of the requirements of internship from April to September

in the

Network and Products Security
Orange Labs. in Caen

September 2014

Contents

List of Figures

List of Tables

Abbreviations

VMI	**V**irtual **M**achine **I**ntrospection
FMA	**F**orensic**M**emory **A**nalysis
SVM	**S**ecure **V**irtual **M**achine
GVM	**G**uest **V**irtual **M**achine
VNI	**V**irtual **N**etwork **I**nterface
IDS	**I**ntrusion **D**etection **S**ystem
QMP	**Q**emu **M**achine **P**rotocol
GPA	**G**uest **P**ysical **A**ddress
DKMS	**D**ynamic **K**ernel **M**odule **S**upport

VM Name	Image Path	OS Type	VMI Application	Login	Password
Win7_64bit	∼/stage/Win7-64bit.img	Win7 SP1 64bit	Nitro, Volatility-LibVMI	win7_64bit	None
vm1	∼/stage/Ubuntu14-02-32.img	Ubuntu14.04 32bit	Volatility-LibVMI	vm1/root	123456
vm2	∼/stage/vm2.img	Ubuntu10.04 32bit	Volatility-LibVMI	vm2/root	123456
vm3	∼/stage/vm3.img	Ubuntu12.04 32bit	Volatility-LibVMI	vm3/root	123456
Win7_32bit	∼/stage/Win7-32bit.img	Win7 SP1 32bit	Volatility-LibVMI	win7_32bit	None
Haiku	∼/virtuoso/haiku-r1alpha2-anyboot.qcow2	Haiku R1 alphaA2	VIRTUOSO	None	None

- For host machine(Workstation HP Z820), its logins and passwords are respectively: "cloud/cloud14" and "root/123456".
- For Windows OS, we have not set a password for any user including "administrator"
- For Ubuntu OS, we have not created password for "root". To be privileged, we rely on "sudo" command with normal user's password
- For Haiku OS, we use an VM image provided by user and load directly its snapshot, thus we do not need user and password

Chapter 1

INTRODUCTION

Virtual machine Introspection, also known as VMI, is an emerging technology largely used in virtual data center in recent years, for the purpose of building a wide range of agentless VM monitored applications such as intrusion detection system [12], virtual firewall [7], etc. According to Pfoh [13], there exist three patterns to help build VMI applications : in-band pattern, out-of-band pattern(used interchangably with term delivery pattern) and derivative pattern. In-band pattern needs an agent installed in monitored guest, thus it is not our investigation focus. Out-of-band is the most popular VMI approach but with some problems of portability. Derivative pattern, relying on hardware architecture information (MMU state, control registers in vCPU, etc.) to infer guest running state, presents a better portability feature and there is no much work in this field. However, as the development in this research field, this classification is not complete to cover all existing VMI technologies. Based on Phof's classification, I suggest adding a new one: reutilization pattern, because recent advance shows that the semantic gap could be largely narrowed by reusing the exercised code from a trusted OS kernel. We plan to make a full and deep state-of-the-art for currently existing VMI technologies, thus in the past six months, we picked a representative VMI applicaitoon (Nitro, Volatility&LibVMI, VIRTUOSO) in each category to install, manipulate and assess in our establised virtualization platform. This article is aimed to record all the exploration process in this road.

Chapter 2

STATE-OF-THE-ART ABOUT VMI

In this section, we firstly present some important background conceptions and terminology about VMI technology, then look through the evolution of VMI, finally analyze and synthetize some typical VMI applications to get a panorama about this emerging technology.

2.1 SEMANTIC GAP [1]

The fundamental challenge faced by all VMI applications is to bridge the semantic gap. Theoretically, the hypervisor is capable of retrieving all low-level binary data (for example memory page, network traffic, etc.) from guests which it manages. These binary data contain almost all running states of VM. However, without more extra semantic knowledge about guest OS or hardware architecture, hypervisor could not extract high-level information about running states of guests and thus is not able to react to guests' activities. This lack of extra semantic knowledge is called semantic gap.

2.2 VMI-RELATED TERMINOLOGY

2.2.1 Semantically Awareness & Semantically Unawareness

According to semantic awareness, all VMI applications fall into two categories: semantically awareness and semantically unawareness [14]. An application characterized as semantic aware means that it conducts monitoring task by extracting information related to guest OS (for example, guest OS kernel data structure, guest memory lay out, etc.). On the contrary, semantic unawareness VMI application seeks to infer other information than OS-related semantic information, a typical example for this kind of application is Antfarm [15]. Obviously, the semantically unaware applications are less powerful compared to semantically aware VMI application in terms of account of running state. However, usually, the former is more robust against a majority of circumstance technique. Thus in practice, it's difficult and usefulness to judge which kind of approach is much better. It's better to combine respective advantages of these two approaches.

2.2.2 Derivative Pattern & Delivery Pattern & Reutilization Pattern

The taxonomy of VMI application could be talked in another perspective. On the basis of the manner by which semantic information is gathered by hypervisor, all VMI applications could be implemented in three patterns: in-band pattern, delivery pattern and derivative pattern [13]. In-band pattern describes the case where an internal agent in VM gather and delivery information to hypervisor. Strictly speaking, In-band pattern is not a real VMI method, thus it is not an investigation focus. Delivery pattern is the most commonly used method to bridge the semantic gap. Its main idea lies in that hypervisor obtains in advance some semantic knowledge, such as System symbol file for Linux guest, to extract subsequently more running states of VM. VMI in delivery is the most active research domain compared to the other two patterns. The last pattern is derivative pattern whose principle is to extract and infer running states information about guest from semantic knowledge of hardware architecture. It seems that this pattern is ideal for implementing VMI applications. However, derivative pattern could be used by nature in limited cases. To what extent this limitation exists is further explored in chapter 6 by Pfoh [16]. In addition, the investigation about derivative pattern is much fewer than delivery pattern. Nitro, Antfarm and Lycosid [15, 17, 18] are three typical VMI applications in this domain. In terms of reutlization pattern, this is a term I personally add to cover those VMI applications whose principal philosophy is reutilization of binary code or execution context [16, 19]. Compared with other patterns (delivery or derivative pattern), it is a big step in automating of introspection tool generation, because it's much more suited in provider cloud environment.

2.3 VMI TECHNOLOGY EVOLUTION

The prevalence of virtualization technologies gives new opportunities for VMI, at the level of hypervisor, to inspect and analyze both the user level program and OS kernel states outside the virtual machine itself. However, the key challenge in VMI is to bridge the semantic gap. Many approaches are proposed to address this problem over the past decade. This section is devoted to talk about the evolution of VMI in terms of implementation philosophy and their respective point of innovation and limitation.

The first attempt [12] of bridging the semantic gap is to leverage the Linux crash utility (a kernel dump analysis tool), but this solution requires the kernel to be recompiled with the debugging symbols. Therefore, limitation of this approach is rather obvious: not convenient and is uniquely applied to open-source OS such as Linux. The significance of this solution resides in that it for the first time validates the practicability of VMI technologies. Much more efforts are still required to make VMI a practical solution in cloud environment.

Given that the view of hypervisor for guest is just raw data (0/1 bit series) of memory and vCPU registers, it is rather intuitive and logic to overcome the semantic gap by leveraging manual kernel data traversal approach. Some research efforts [7] are representative examples following this approach. Its idea is to locate, traverse, and interpret known data structure of the in-guest memory. While this solution has been widely adopted between 2003 and 2011, many of them rely on a manual effort or compiler-assistant approach to locate the in-guest kernel data and develop the in-guest semantic-equivalent code for the introspection. This approach is indeed a significant step in the road to remedy the semantic gap. However, VMI introspection applications based on this approach suffer from frequent changes due to the constant update or patch of kernel. More importantly, this approach requires an intimate knowledge about OS kernel or reverse engineering.

Furthermore, it may also introduce vulnerabilities for attackers to evade these hand-built introspection tools.

To relieve the painful process of adapting introspection utilities caused by the changes to OS kernel, some researchers [15, 16] tried to retrieve meaning high-level information from hardware architecture such as Intel X86 architecture. Although this method is guest-OS agnostic, the high-level information could be revealed is rather limited. Thus this approach is also far from practical for cloud providers.

All existing solutions to narrow semantic gap before 2011 mostly rely on manual efforts and reverse engineering skills, which pose problems to deploy VMI in cloud environment. Even though we have already a perfect approach to bridge the semantic gap, we still have to develop some guest OS management utilities from scratch to mimic the similar in-guest inspection programs (ps/netstat,etc.). Based on this observation, people think of reusing the legacy binary code instead of developing extra new vulnerability-prone programs. Under the guide of this philosophy, Dolan-Gavitt et al [11] presented VIRTUOSO, which made a significant attempt in this perspective. VIRTUOSO made a first step showing that we could actually reuse the legacy binary code to automatically create VMI tools with the assistance from a human expert. Its key idea is to first train each in-VM program (e.g., ps) and then translate the trained traces (essentially slices) into an independent introspection program running at the hypervisor layer. However, due to the nature of dynamic analysis, VIRTUOSO is only able to reproduce introspection code that has been executed and trained. Meanwhile, it is not fully automated and requires the intervention from a human expert.

Drawing inspiration from VIRTUOSO, VMST [19] shows a dual-VM based online kernel data redirection approach that addresses the limitations from the training existing in VIRTUOSO. Unlike VIRTUOSO, VMST reuses the execution context of an inspection process in a trusted VM: When a kernel instruction accesses the kernel data of introspection interest, it redirects the data from the guest VM to the trusted VM. Of course, VMST has its own limitations : Firstly, VMST is not entirely transparent to abitary OS kernel(In fact, No VMI technology has this ability.) and it still binds with some particular OS kernel knowledge such as sytem call interface, interrupt handling and context switching,etc. Second, VMST does not support asynchronous system calls.

Third, VMST may encounter some issues when guest OS runnin in multi-CPUs architecture.

Built atop of VMST, the same research team proposed EXTERIOR [20], which is a novel dual-VM based external shell for trusted, native, out-of-VM program execution for guest-OS administration including introspection, configuration and recovery. Unlike preexisting VMI techniques, EXTERIOR for the first time enables an out-of-VM shell with a guest-OS writable and executable capability. Its key idea is to leverage an identical trusted kernel with the guest-OS to create the necessary environment for a running process in a SVM (trusted VM), and transparently and dynamically redirect and update the memory at the VMM level to a GVM, thereby achieving the same effect in terms of kernel state updates of running a program inside a guest-OS. Traditionally, shells are designed atop an OS kernel, EXTERIOR demonstrates that a shell could also be designed below an OS. Thus, one important significance of EXTERIRO lies in that it presents a new program execution model on top of virtualization. While EXTERIOR has made an early attempt of building a hypervisor layer shell, it has a lot of constraints and is far from practical: Firstly, it has to first perform the guest OS fingerprinting [21] and then use the exact same version of the guest OS running in a SVM to introspect the kernel state of a GVM. Second, it can suffer from various failures and shortfalls when an introspection related system call uses kernel synchronization primitives [19]. Third, it is built atop a binary code translation based VM which often has 10-40X performance slowdown.

HyperShell [22] has drawn inspirations from Process Implanting [23] and extended EXTERIOR. To overcome the semantic gap challenge, HyperShell introduces a reverse system call(R-syscall in short) abstraction. This abstraction serves as the interface in a reverse direction from a layer up and it is also transparent to legacy software (e.g. ps/lsmod/netstat/ls/cp). This design allows a large number of legacy in-guest management utilities executing directly at hypervisor level with no modification. HyperShell still remains some limitations: First, it needs a log record at the hypervisor layer for each activity executed in HyperShell (Security concern). Second HyperShell requires a trusted guest OS kernel and init process thus could not be used for security critical applications. Third, current prototype requires both OS running in the host OS and the GVM to have compatible syscall. Finally, static linked native utilities cannot be executed in HyperShell. It seems that HyperShell is an attractive propostion for cloud service provider if it is enough mature. Thus, this is an idea worth much attention.

Briefly speaking, until now, there does not exists a VMI technology applicable in real production environment, and the semantic gap problem is still a difficult, open research problem today. More advances and innovations are required to make VMI technology applicable in production environment.

2.4 REPRESENTIVE VMI APPLICATION

As one of internship's objectives, plenty of VMI applications or development frameworks are analyzed and synthesized. It is at the same time an interesting complement for above section in another perspective. Because of space constraints, it's impossible to collect and analyze all existing VMI applications. Here, we just pick up those applications regarded as milestone and having reference value in this domain.

2.4.1 Delivery Pattern VMI Application

VMI theory is initially proposed by Garnkel and Rosenblum [12] in 2003. As a proof-of-concept prototype of their theory, Lirewire [12] is the first VMI application whose objective is to deploy a firewall on hypervisor level. The significance of Lirewire is to prove the feasibility of VMI from within hypervisor. However, it could not give us technical inspiration, thus here its implementation will not be talked about.

VIX [2], which stands for Virtual Introspection for Xen, is a tool suite on the basis of the out-of-band delivery pattern for forensic analysis. This system uses delivered knowledge about the guest OS in order to implement common UNIX tools such as ps, lsmod and netstat. Although this tool suite is close to what we want to implement, it unfortunately does not support KVM hypervisor and its code is not accessible.

RTKDSM [14], for Real-Time Kernel Data Structure Monitoring, leverages the rich OS analysis capabilities of Volatility [4] to significantly simplify and automate analysis of VM execution states. Its implementation is under Xen platform, inspiration that it could give us is the use of Volatility who provides vast kernel data structure knowledge to remove the burden of bridging semantic gaps from VMI application developers.

Other systems such as HookSafe, NICKLE are all presented by Pfoh in his work [16]. They both rely on out-of-band to mitigate the semantic gap to monitor or protect the integrity of kernel code. As their close source nature, they could not give us more help. All mentioned VMI applications are summarized in the Table 2.1. It is supposed to note that this is not an exhaustive table and there all still many others VMI applications in this filed.

TABLE 2.1: Out-of-Band Pattern VMI Applications Non-exhaustive Table

VMI Application	Hypervisor	Open-Source	Features
Livewire [12]	VMWare workstation	No	A VMM-based IDS prototype implementation using VMI technology
VIX [2]	Xen	No	Unix-like utility suit(ps/netstat/lsmod) for generic forensic analysis
HookSafe,NICKLE [24]	Xen	No	Monitor or protect the integrity of kernel code or data structure
Rhee et al's system [25]	QEMU	No	Monitor or protect the integrity of kernel code or data structure
RTKDSM [14]	Xen	No	With the power of Volatility, RTKDSM is an extension software framework meant to be extended to perform application-specific VM state analysis. It supports real-time monitoring of any changes made to the extracted OS states of guest VMs.

2.4.2 Derivative Pattern VMI APPLICATION

Then we talk about some VMI applications based on derivative pattern. Manitou system use x6's paging mechanism to perform integrity checks before code execution. From this work, we find paging mechanism is an important filed to explore to leverage derivative method. Antfarm is another typical example for derivative pattern. It is specially designed to monitor the VM's memory management unit (MMU). From that, it could construct the virtual-to-physical memory mapping and retrieve all running processes identified by a CR3 register value in guest. Lycosid allows detecting and identifying all those hidden processes in guest. It first retrieves two different level process lists (cross-view validation) in guest by respectively using Antfarm and guest built-in utility (for example "ps" command in Linux/Unix world.). If number of processes contained in these two different lists is not pertinent, Lycosid then infer the presence of some hidden processes and deduce those processes hidden by some statistical methods. Antfarm and Lycosid are both implemented in Xen platform and we have no access to its source code. However, these two systems show a typical situation where we could apply derivative pattern. From these two applications, we also get to know that derivative pattern is really useful for security purpose, due to the fact that even though Lycosid is not familiar with the malicious process (PID, process name, etc.), it could interfere those dangerous activities.

Nitro, as far as I know, is the most recent derivate-pattern VMI application, whose objective is to trap system call events according to user-defined filter rule. For example, Nitro could be used as a technical building block retrieving all system calls for malware analysis. Due to its derivative pattern and open-source nature, Nitro is the focus of my investigation. Based on its functionality to access vCPU's registers and system call trap, we intend to add network traffic monitoring functionality to Nitro and implement network monitoring on process granularity. However, Nitro is just under maintenance of Pfoh and has no updates since almost six months, even though it is an open-source project. Thus, there is not enough documentation about its utilization and it needs some enhancements besides its provided basic functionality. Further introduction about Nitro are in the following part. All these derivative pattern VMI applications are summarized in Table 2.2.

2.5 VMI RELATED TOOLS

Besides those mentioned VMI applications, we then talk about some interesting VMI Tools or development frameworks. These tools themselves are not VMI applications aiming for certain security problems. Instead they may be in form of C library providing general VMI functionalities or forensic analysis frameworks to help bridge the semantic gap. All the three frameworks talked are resumed in Table 2.3

2.5.1 LibVMI/XenAccess [2, 3]

LibVMI is an introspection library focused on reading and writing memory from running virtual machines. For convenience, LibVMI also provides functions for accessing CPU registers, pausing and unpausing a VM, printing binary data, and more. LibVMI is designed to work across multiple virtualization platforms. LibVMI currently supports VMs running in either Xen or KVM. LibVMI also supports reading physical memory snapshots when saved as a file.

XenAccess is predecessor of LibVMI but is focused exclusively on Xen. LibVMI is designed to work across a variety of virtualization platforms including Xen and KVM. LibVMI provides a more intuitive API and therefore is more advised compared to XenAccess. Please refer to the following link for more information: https://code.google.com/p/vmitools/.

2.5.2 Volatility [4]

The Volatility is an excellent open-source forensic analysis framework implemented in Python. It presents the advantage of bridging the semantic gap regardless of the system being monitored. Volatility has a wide range memory dump support, from Windows to Linux, from Mac OS to Android. Also, there exist plenty of tutorials about this framework. Thus, recent years have witnessed increasing adoption of Volatility in VMI

Table 2.2: Derivative Pattern VMI Applications Non-exhaustive Table

VMI Application	Hypervisor	Open-Source	Features
Manitou [26]	Xen	No	It uses the paging mechanism of x86 processors from Intel and AMD to perform integrity checks on code pages before execution
AntFarm [15]	Xen	No	It makes use of the paging mechanism and the memory management unit(MMU) in x86 and SPARC hardware architecture to identify running process
Lycosid [18]	Xen	No	Lycosid is capable of detecting and identifying those hidden process on the basis of AntFarm
Nitro [17]	KVM	Yes	Now Nitro supports guest vCPU register access, trap syscall/sysret It works under 64-bit Linux host machine(tested on Ubuntu13.10 with kernel version 3.11 and 3.13). It supports only 64-bit Intel processor with virtualization Extension(VMX)

application development, for example RTKDSM system. For more information, refer to this link: https://code.google.com/p/volatility/.

2.5.3 Insight-VMI [5, 6]

Similar to Volatility, Insight-VMI is another open-source forensic analysis framework written in C++. Compared to Volatility, it additionally provides an interactive shell for manual analysis of kernel objects and a JavaScript engine for automated analysis of repeating or complex tasks, but it now supports only Linux memory analysis. In addition, Insight-VMI and Nitro are both developed and maintained by the same study team at the Technische Universität München in Germany. Detailed information is available in: https://code.google.com/p/insight-vmi/.

Table 2.3: VMI Tool Manifest

VMI Tool	Hypervisor	Open-Source	Features
LibVMI [1]	KVM, Xen	https://code.google.com/p/vmitools	• Works with 32-bit and 64-bit Windows and Linux guests (64-bit support in version 0.8 and newer) • Works with physical memory snapshots saved in a file (e.g., VMWare snapshots) • Pause/unpause the VM through an API function • Volatility address space plugin enabled running Volatility on a live VM
Volatility [27]	KVM,Xen	https://code.google.com/p/volatility	• supports forensic analysis of the Windows,Linux,Mac OS memory images • Does not provide memory sample acquisition • Supports a variety of sample file formats
Insight-VMI [28]	KVM,Xen	https://code.google.com/p/insight-vmi	• Analysis of physical memory dumps or live physical memory of a virtual machine • Support for 32-bit and 64-bit addressing schemes with and without PAE • Re-creation of kernel objects for the Linux kernel • Automatic de-referencing of pointers to futher objects • Application of type casts and pointer arithmetic as the kernel would do • An interactive shell for manual analysis of kernel objects • A JavaScript engine for automated analysis of repeating or complex task

Chapter 3

INTERNSHIP'S OBJECTIVE

As mentioned before, in terms of bridging semantic gap for VMI, the most popular method is delivery pattern. In this field, there exist plenty of VMI applications for various purposes. Although this method could help to effectively mitigate the semantic gap, it poses other problems such as portability, non-robust against circumstances technologies, etc. For example, if we use LibVMI to help parse guest Linux kernel data structure, the system symbol file is required firstly to be transferred to host machine. In case of kernel's update in guest, this file transfer needs to be executed again. In this context, we want to explore the potential of derivative pattern, which relies on guest hardware architecture to extract useful information. Pfoh has declared in his work [16] that:

"These methods allow one to enumerate the running processes, monitor system calls, or track network connections on a per-process basis in a completely guest OS agnostic manner."

His declaration is rather interesting and attractive even though Pfoh has not given more explanation to argue his idea. Thus, the main objective of this internship is to prove this idea and develop a prototype implementation. The prototype should present the following features:

- Capable of tracking network connection on a per-process
- Works in derivative method and independent of guest OS

In conclusion, it's a "netstat-like" utility by leveraging derivative pattern to bridge semantic gap and works out of monitored guest.

We plan to achieve our determined goal by the following steps:

- Install and manipulate Nitro to see how derivative pattern works
- Study which component in KVM virtualization platform is in charge of virtual networking
- Study how to enumerate running process in a guest agnostic manner
- Study how to correlate each network connection with identified running process

Chapter 4

STUDY OF VMWALL[7]

Our strategy to achieve our goal is firstly to absorb inspiration from already-existing VMI applications. Among all the VMI applications that we have studied, VMWall is relatively much closer to our design goal. This system implements an application-level firewall working in Xen hypervisor. To achieve this, it needs to correlate each monitored TCP or UDP connection with process which creates it. Although VMWall is implemented in Xen hypervisor and its source code is not accessible, its work about how to correlate network traffic and corresponding process may give us some clues.

VMWall consists of two parts: user agent and kernel component. Kernel component uses a modified ebtables [29] packet filter to intercept all packets sent to or from a guest domain. To well understand VMWall's kernel component implementation, it is supposed to be familiar with Xen network virtualization. In fact, Xen offers several different 9networking modes. VMWall uniquely considers the bridging mode. With this mode, Dom0 provides a virtual Ethernet bridge connecting the physical network card to all virtual network devices provided by Xen to the domU VMs. Dom0 uses its virtual bridge to multiplex and demultiplex packets between the physical network interface and each unprivileged virtual machine's VNI (Virtual Network Interface). Due to this fact mentioned, kernel component (in this case Ebtables) could intercept all packets between monitored VM and external Internet and retrieve information like IP address and TCP/UPD port to identify a network connection.

If necessary, user agent, another important part of VMWall, will receive a request containing IP address and TCP ports information and need to use the latter to identify the corresponding process running in monitored guest by out-of-band pattern VMI. The correlation between network connection and process relies on one file type in Linux/Unix: socket. Socket is a special file type in Linux/Unix world. To create a socket connection, IP address and transport layer port are required. All opening port numbers in Linux are hashed and managed by data structure "inet_hashinfo". By iterating this variable (for example tcp_hashinfo) of this data structure, the socket file using a given port number could be identified. Similarly, all processes in Linux are managed by a double linked data structure "task_struct". With input of a certain socket reference, the process creating this socket could be found. The Figure 4.1 shows how to correlate network connection and process on data structure level.

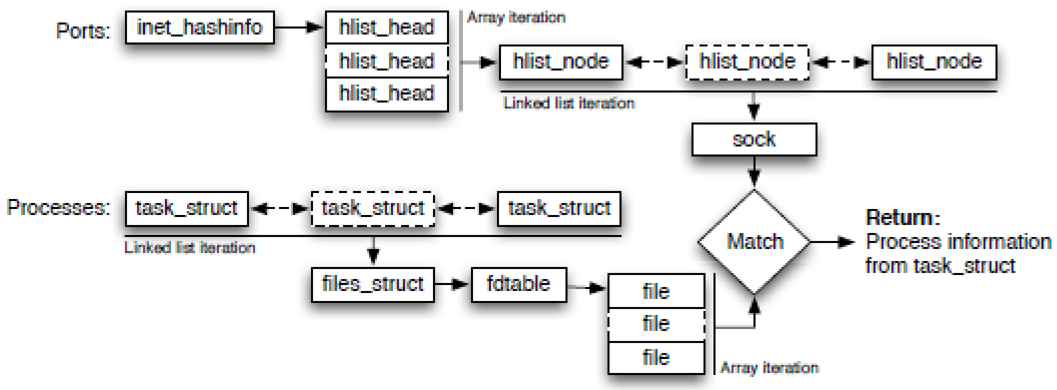


FIGURE 4.1: Guest Linux kernel data structures traversed by the VMWall user agent during correlation of the process and TCP packet information [7]

Chapter 5

ESTABLISHMENT OF EXPERIMENTATION PLATFORM

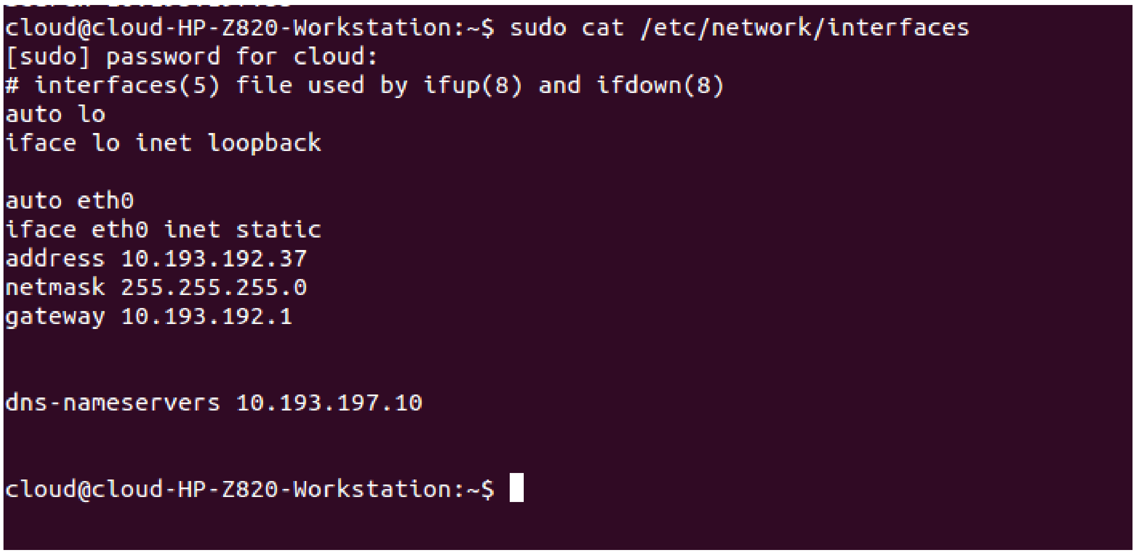
As the state-of-the-are about VMI is finished and the objective is determined, we plan to implement a prototype in KVM virtualization platform. With a HP workstation at disposal, we need firstly to establish an experimentation and development platform.

5.1 WIN7 and UBUNTU DUAL-BOOT SYSTEM

As was mentioned above, the first step to achieve our final objective is to figure out how derivative pattern functions by installing and manipulating the only known open-source derivative pattern VMI application: Nitro. Thus, it's logical to choose those operating systems which support Nitro. According to Pfoh, Nitro has been tested on Linux kernel 3.11 and 3.13. Hence, Ubuntu14.04 with kernel version 3.13 is our final choice. KVM virtualization platform will be built in a HP Z820 workstation which initially has only Windows 7 installed. The first step is to install a Windows 7 and Ubuntu 14.04 dual-boot system. This manipulation is not difficult but kind of tedious. In fact, we follow this tutorial http://askubuntu.com/questions/343268/how-to-use-manual-partitioning-during- installation to accomplish this step.

5.2 NETWORK CONFIGURATION

This workstation has been allocated a static network configuration and it uses always port 8 in each office. To achieve this, we need to edit configuration file /etc/network/interfaces to set 10.193.192.37 as ipv4 address, 255.255.255.0 as 10network mask, 10.193.192.1 as default gateway and 10.193.197.10 as DNS name server. The final network configuration in Ubuntu14.04 is shown as in Figure 5.1.



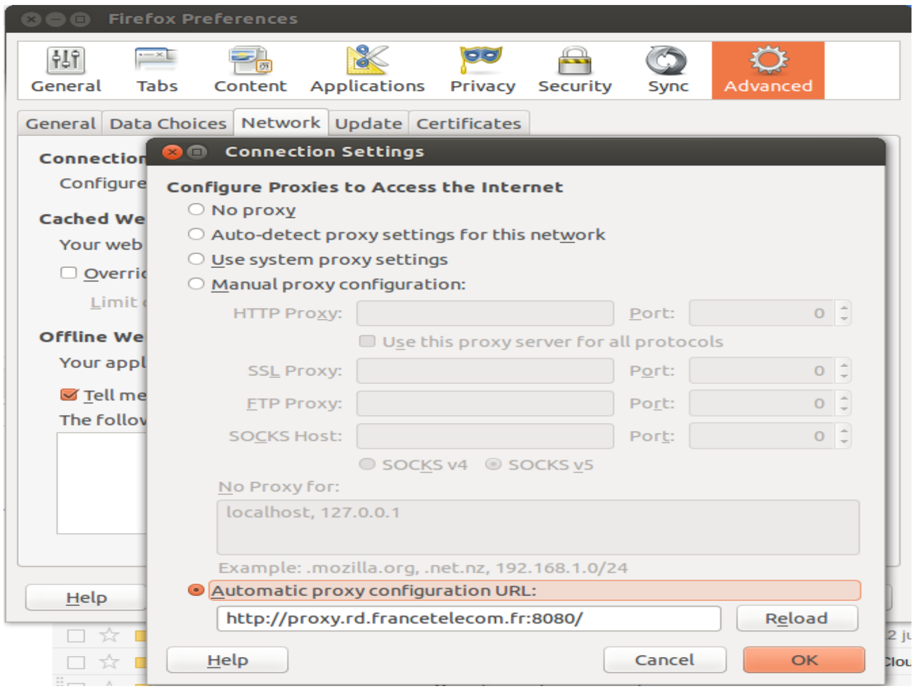
```
cloud@cloud-HP-Z820-Workstation:~$ sudo cat /etc/network/interfaces
[sudo] password for cloud:
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 10.193.192.37
netmask 255.255.255.0
gateway 10.193.192.1


dns-nameservers 10.193.197.10


cloud@cloud-HP-Z820-Workstation:~$
```

FIGURE 5.1: HP Workstation static network configuration

Besides, never forget to set a parameter named "managed" to "true", defined in configuration file /etc/NetworkManager/NetworkManager.conf. Otherwise the network will be always in disable state. The security politics of Orange Labs require each PC to configure proxy server if the former needs Internet connectivity. In fact, under Linux system, it is possible to set proxy configuration on different level. The first solution is to declare http proxy server on command-line level. For example, if we want to update APT source list, we could issue a command in a terminal:

```
# sudo http://proxy.rd.francetelecom.fr:8080 apt-get update
```

This proxy configuration is uniquely valid for the current command and every command we tape should contain an option for proxy server, which is not effective and tedious. Proxy server configuration could also be set on application level. For example, if we want to user web browser such as Firefox, we need to set proxy address for Firefox just like in the following picture.



Connection Settings

Configure Proxies to Access the Internet

- ○ No proxy
- ○ Auto-detect proxy settings for this network
- ○ Use system proxy settings
- ○ Manual proxy configuration:

HTTP Proxy: Port: 0

☐ Use this proxy server for all protocols

SSL Proxy: Port: 0

FTP Proxy: Port: 0

SOCKS Host: Port: 0

○ SOCKS v4 ● SOCKS v5

No Proxy for:

localhost, 127.0.0.1

Example: .mozilla.org, .net.nz, 192.168.1.0/24

- ● Automatic proxy configuration URL:

http://proxy.rd.francetelecom.fr:8080/ Reload

Help Cancel OK

FIGURE 5.2: Firefox proxy parameter

To use APT command to install software, we need to edit (if not exist, create) a configure file named ‘apt.conf ’under /etc/apt. This file just contains one line: Acquire::http::proxy::”http://proxy.rd.francetelecom.fr:8080” Then we could issue command like ‘sudo apt-get install software_name ‘as normal in terminal. In one word, it is supposed to configure proxy server address for every application which needs Internet connectivity. The last solution is setting bash configuration files like /etc/profile, /etc/bash.bashrc or /.bashrc. Proxy configuration in this manner is user or all-user level. Only once configuration is needed to guarantee Internet connectivity for all applications.

5.3 KVM Introduction

After installation of Ubuntu14.04 in our workstation, it’s time to install required Ubuntu packages to turn Ubuntu into a hypervisor. Prior to the real installation, it is necessary to make a presentation for KVM.

KVM (Kernel-based Virtual Machine) is a virtualization infrastructure for the Linux kernel that turns it into a hypervisor, which was merged into the Linux kernel mainline in February 2007. KVM requires a processor with hardware virtualization extension. KVM has also been ported to FreeBSD and Illumos in the form of loadable kernel modules. Compared to other virtualization alternatives available in Linux, KVM presents the following advantages [8]:

- Can interact directly with the Kernel
- Default virtualization in leading Linux Distributions
- One of the Linux software developed aggressively.
- Almost becoming competitor to VMware by implementing technologies such as v2v, p2v, and many open source tools to manage VM
- Number of open source cloud automation softwares use KVM as default hypervisor

5.4 KVM's Operating Principle [8]

Once KVM is installed on a Linux box, a hardware file /dev/kvm is created which will act as interpreter between actual hardware and hypervisor manager (for example, virt-manager). Whenever a request for hardware changes or additions comes from hypervisor manager, KVM software starts allocating those resources virtually by interacting with real hardware. Suppose we want to change RAM on a virtual machine, this is communicated by the hypervisor manager to KVM for allocating the resource. Then KVM interacts with hardware and reserves that RAM from real RAM for that particular VM. This happens for the other resources as well.

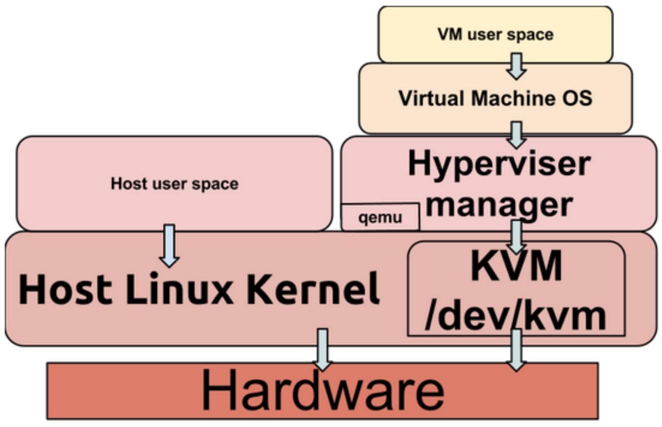


FIGURE 5.3: KVM Virtualization Architecture in Linux [8]

5.5 KVM's VM Management Tools

There exists several methods in KVM for management activity such as guest's creation, deletion, start, stop and clone,etc.

5.5.1 virt-manager

The most popular GUI is called Virtual Machine Manager (VMM), developed by Red-Hat. The tool is also known by its generic package name virt-manager. It comes with a number of supporting tools, including virt-install, virt-clone, virt-image, and virt-viewer, which are used to provision, clone, install, and view virtual machines, respectively. VMM also supports Xen machines. Virt-manager relies on libvirt and uses qemu to run guest instance. In the following section, we will show how to create a KVM guest with virt-manager.

5.5.2 virsh

The generic KVM command interface is provided by virsh. virsh is a shell for managing hypervisors and VM's directly from host OS terminal. Specifically, you can use the supporting tools, like virt-install for creating your virtual machines. On Ubuntu, there's a special ubuntu-vm-builder tool that can be used for provisioning Ubuntu builds, developed by Canonical.

5.5.3 qemu command line with "enable-kvm" option

In fact, virt-manager or virsh both could be regarded as wrapper based on libvirt and qemu. As a result, we could use qemu command lines to manage KVM guests with "enable-kvm" option to profit performance acceleration offered by KVM. However, this method is not handy compared to virt-manager and virsh. Note that Nitro is only able to monitor KVM guests launched by a modified version qemu. We could use GUI tools such as virt-manager to create virtual machines images and issue qemu command line to start created VM and start Nitro to monitor these VMs.

5.5.4 KVM command

KVM also has its own syntax, similar to QEMU. It is not a recommended way of managing virtual machines. More information is available with link: https://help.ubuntu.com/community/KV

5.5.5 Libvirt

Libvirt is not a VM management tool. Instead, it is a toolkit to interact with the virtualization capabilities of recent versions of Linux [30]. It is an open-source project and provides a set of long term stable C API. These C APIs all APIs are designed to do virtualization management, such as: provision, create, modify, monitor, control, migrate and stop the guests - within the limits of the support of the hypervisor for those operations. Hence, libvirt is intended to be a building block for higher level management tools and for applications focusing on virtualization of a single physical host. The Figure 5.4 shows the relationship between hypervisor, libvirt and virtualization management tool.

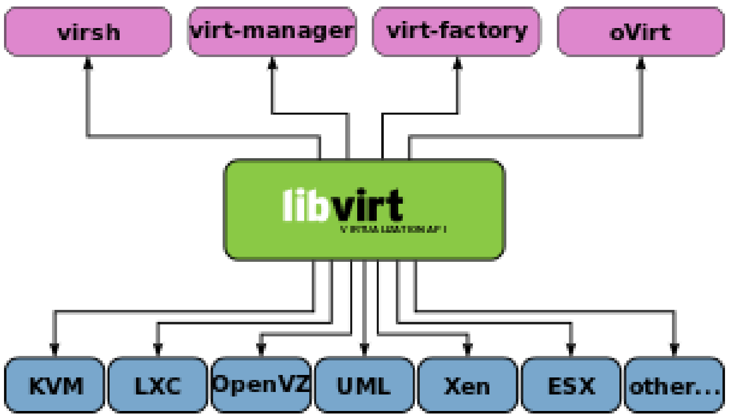


FIGURE 5.4: Relationship between virsh, virt-manager and libvirt [31]

5.6 KVM/QEMU networking

Guest (VM) networking in KVM is the same as in qemu, so it is possible to refer to other documentations about networking for qemu. In this section, we will talk about the three most frequent types of network needed.

5.6.1 User Networking

This networking mode, shown in figure 5.5, allows guests accessing to the host, to the internet. However, the guests are neither invisible from outside network nor from other VMs. In addition, user networking does not support other network protocols other than TCP/UDP. Hence, certain applications (like ping) won't work.

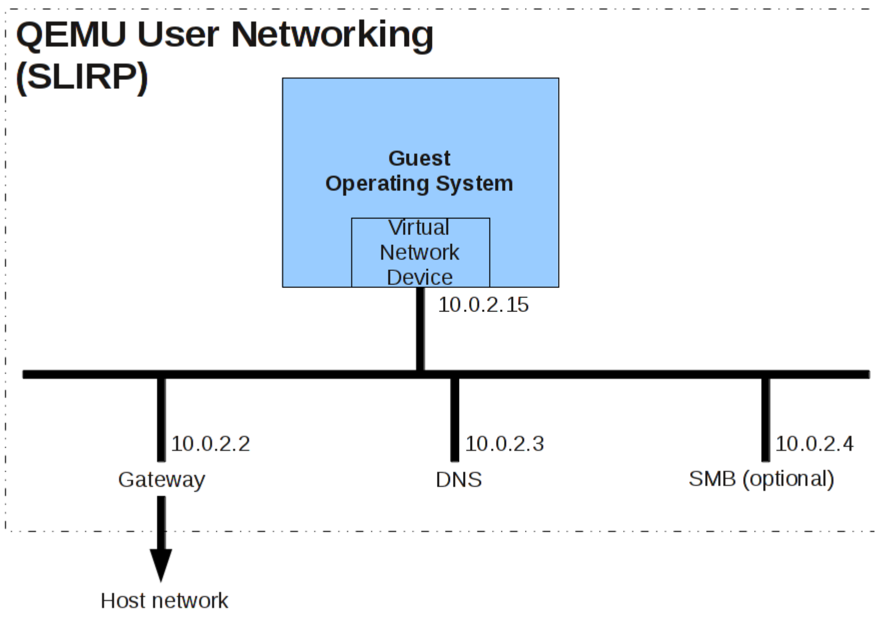


FIGURE 5.5: User networking mode topology [32]

For example, we could issue the following to start a guest in user networking mode:

```
# qemu-system-x86_64 -enable-kvm -had win7.img -m 2048
```

The guest OS will see an E1000 NIC with a virtual DHCP server on 10.0.2.2 and will be allocated an address starting from 10.0.2.15. A virtual DNS server will be accessible on 10.0.2.3.

5.6.2 Bridged Networking

The bridge networking mode makes all launched guests run just like they are in the same local network with host machine. To use bridged networking, a virtual Ethernet bridge should be firstly created. Under Ubunu14.04, we need to edit /etc/network/interfaces as following, for example:

```
# The content of file /etc/network/interfaces
# Replace old eth0 config with br0, there no more "auto eth0"
```

```
auto br0
# Use old eth0 config for br0, plus bridge stuff
iface br0 inet static
# Attention use old static configuration of eth0
bridge_ports eth0
bridge_stp off
bridge_maxwait 0
bridge_fd 0
```

Run each guest with the following, replacing $macaddress with a customized MAC address.

```
# qemu-system-x86_64 -hda /path/to/hda.img -device e1000,netdev=net0,mac=$macaddress-netdev
tap,id=net0
```

5.6.3 networking mode

NAT networking mode is actually the default networking mode for virtual machines created by virt-manager.

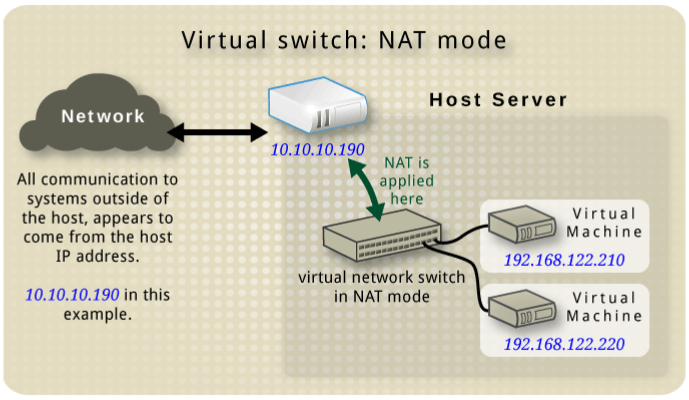

Virtual switch: NAT mode
Network
Host Server
10.10.10.190
NAT is applied here
Virtual Machine
192.168.122.210
Virtual Machine
192.168.122.220
virtual network switch in NAT mode
All communication to systems outside of the host, appears to come from the host IP address.
10.10.10.190 in this example.

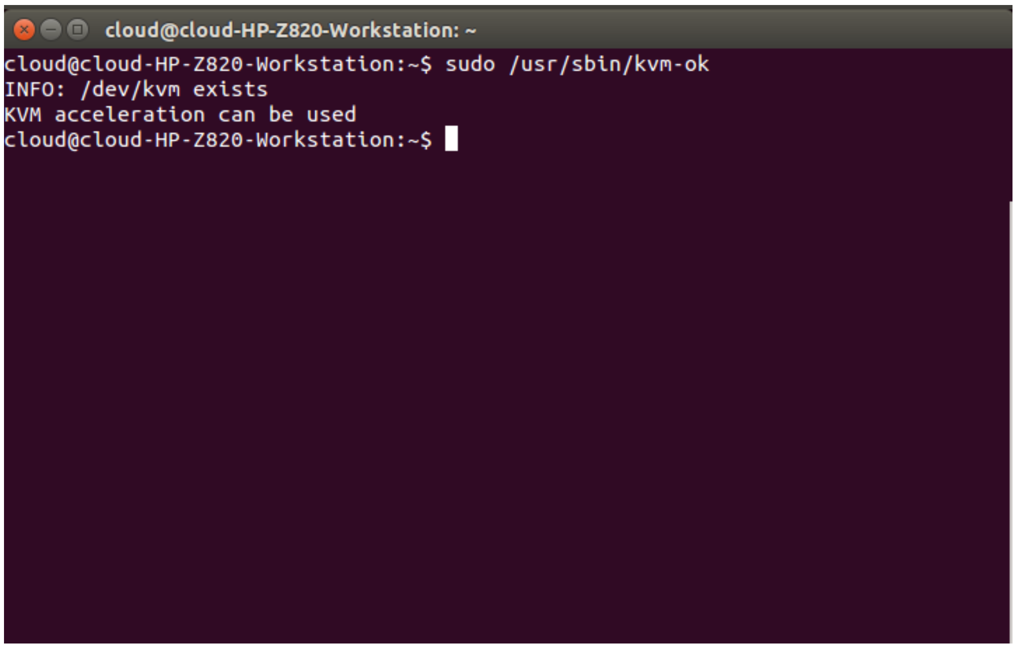
FIGURE 5.6: Virtual network on NAT mode [33]

5.7 KVM installation

KVM is a virtualization feature in the Linux kernel that lets a program like qemu safely execute guest code directly on the host CPU. This is only possible when the target architecture is supported by the host CPU. CPU's virtualization extension (VMX for Intel's processors and SVM for AMD.) support could be verified by issuing this following command in a terminal:

```
#  grep -c '(vmx|svm)' /proc/cpuinfo
```

A return value greater than zero means that current CPU supports KVM. In addition, it's necessary to check virtualization technology is enabled in BIOS. After enabling this feature, we have to cold power-cycle the machine for the change to take effect. Once this is done, run kvm-ok to verify:



```
cloud@cloud-HP-Z820-Workstation:~$ sudo /usr/sbin/kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
cloud@cloud-HP-Z820-Workstation:~$ 
```

Figure 5.7: Output of kvm-ok command

Then we begin to install required packages for KVM. The command to install everything you need is:

```
#  sudo apt-get install qemu-kvm libvirt-bin ubuntu-vm-builder bridge-utils
# sudo apt-get install virt-manager virtinst qemu-system
```

Out of interest, KVM doesn't have its own configuration directory. The configuration files could be found in: /etc/libvirt/qemu/ There are some quirks or confusions which deserve clarify, in term of difference and association between libvirt, qemu, qemu-kvm.

Qemu in fact is project independent of KVM, it is an emulator which itself could be used as a virtualization alternative. Qemu-kvm is fork of QEMU maintained by KVM project. Currently (close to the 1.1 release) it still provides the best performance and certain additional features for using KVM with QEMU on x86. Any other architecture is already fully supported by QEMU itself. However QEMU development community plans to suspend the development of qemu-kvm and make qemu master fork to support all architectures. Libvirt is a toolkit to interact with the virtualization capabilities of recent versions of Linux [23]. It is an open-source project and provides a set of long term stable C API. These C APIs all APIs are designed to do virtualization management, such as: provision, create, modify, monitor, control, migrate and stop the guests - within the limits of the support of the hypervisor for those operations. Hence, libvirt is intended to be a building block for higher level management tools and for applications focusing on virtualization of a single physical host. The following picture shows the relationship between hypervisor, libvirt and virtualization management tool.

5.8 KVM Win7 Guest Manipulation

After establishing KVM virtualization platform, we plan to create a Windows 7 64 bits guest with virt-manager. To launch virt-manager, open a terminal and issue the following command:

```
#  sudo virt-manager
```

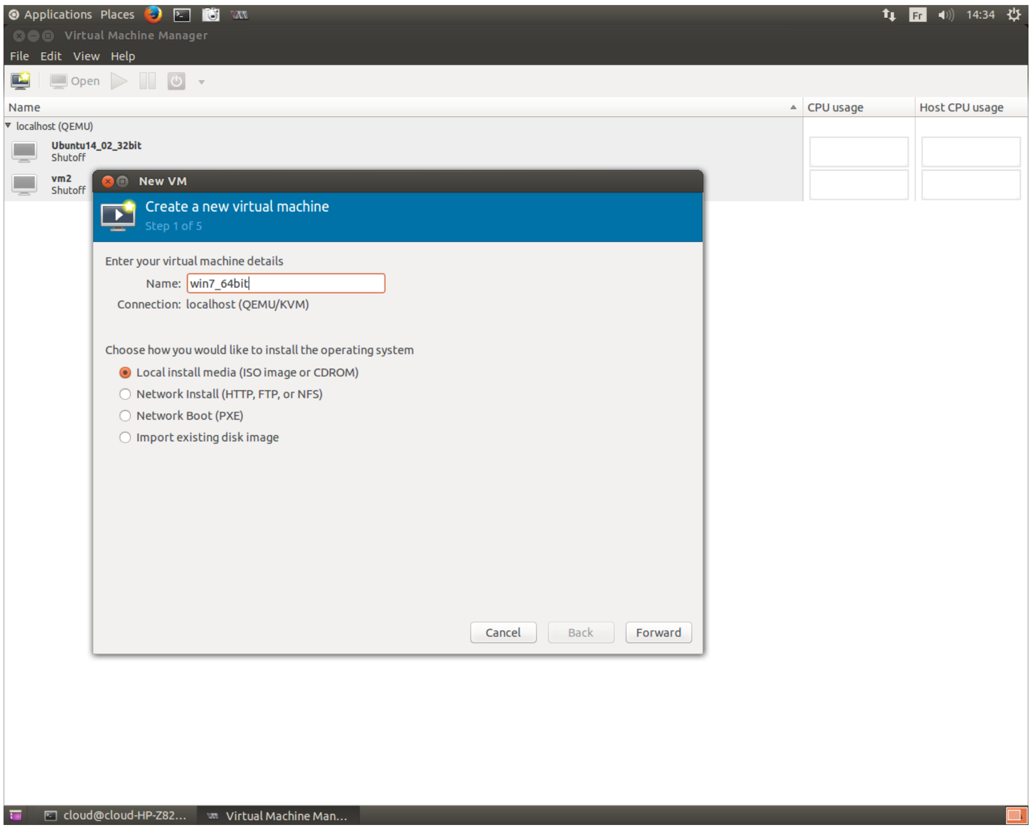
We name this guest as “win7_64bit”, chose a local installation place then click forward button.

At step 2 shown in Figure 5.9, we need to indicate the location of the win7 installation ISO file for virt-manager.

Then we create a virtual disk of 20G for our guest in step 3 (Figure 5.10).

In the next step (Figure 5.11), our reserved virtual disk is used as hard disk for win7_64bit guest.

The last step (Figure 5.12) is a resume of installation configuration. Note that the guest works with NAT network mode.



Create a new virtual machine
Step 1 of 5

Enter your virtual machine details

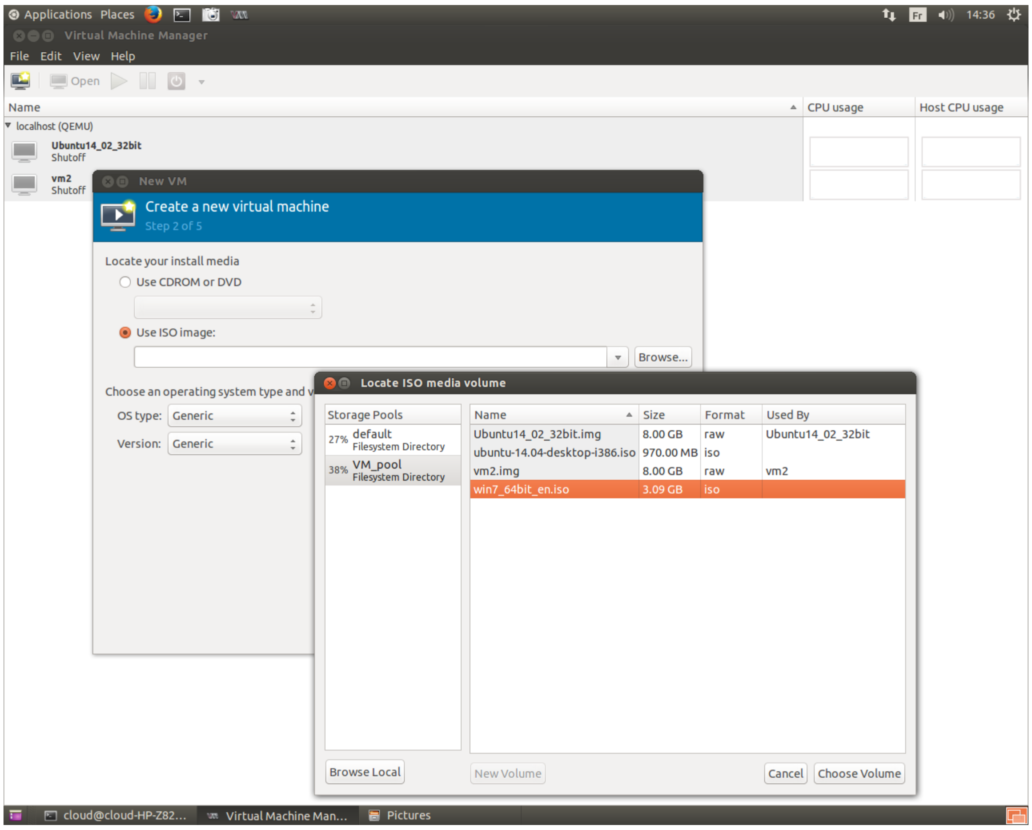
Name: win7_64bit

Connection: localhost (QEMU/KVM)

Choose how you would like to install the operating system

- [✓ Local install media (ISO image or CDROM)]
- Network Install (HTTP, FTP, or NFS)
- Network Boot (PXE)
- Import existing disk image

FIGURE 5.8: Step 1-Create a new VM with name win7_64bit



Create a new virtual machine
Step 2 of 5

Locate your install media

- Use CDROM or DVD
- [✓ Use ISO image:]

Choose an operating system type and v

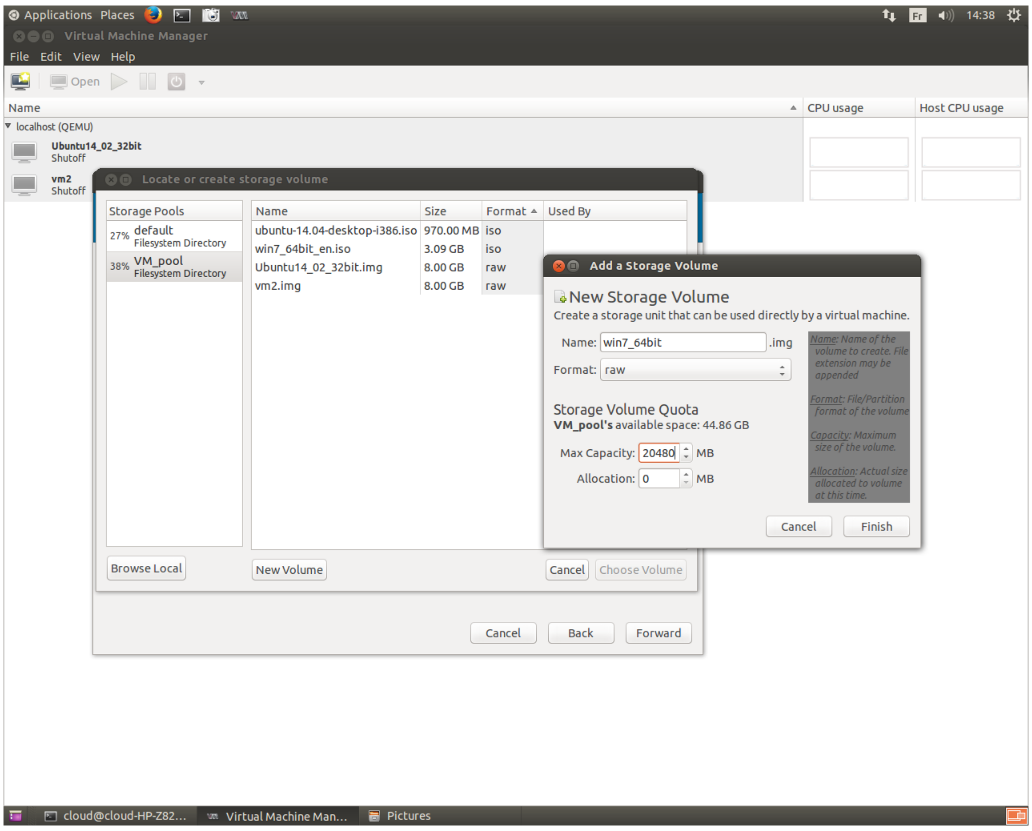
OS type: Generic

Version: Generic

Storage Pools
27% default Filesystem Directory
38% VM_pool Filesystem Directory

Name	Size	Format	Used By
Ubuntu14_02_32bit.img	8.00 GB	raw	Ubuntu14_02_32bit
ubuntu-14.04-desktop-i386.iso	970.00 MB	iso	
vm2.img	8.00 GB	raw	vm2
win7_64bit_en.iso	3.09 GB	iso	

FIGURE 5.9: Step 2-Choose win7 ISO file



Storage Pools	
27%	default Filesystem Directory
38%	VM_pool Filesystem Directory

Name	Size	Format	Used By
ubuntu-14.04-desktop-i386.iso	970.00 MB	iso	
win7_64bit_en.iso	3.09 GB	iso	
Ubuntu14_02_32bit.img	8.00 GB	raw	
vm2.img	8.00 GB	raw	

New Storage Volume

Create a storage unit that can be used directly by a virtual machine.

Name: win7_64bit .img

Format: raw

Storage Volume Quota

VM_pool's available space: 44.86 GB

Max Capacity: 20480 MB

Allocation: 0 MB

Name: Name of the volume to create. File extension may be appended

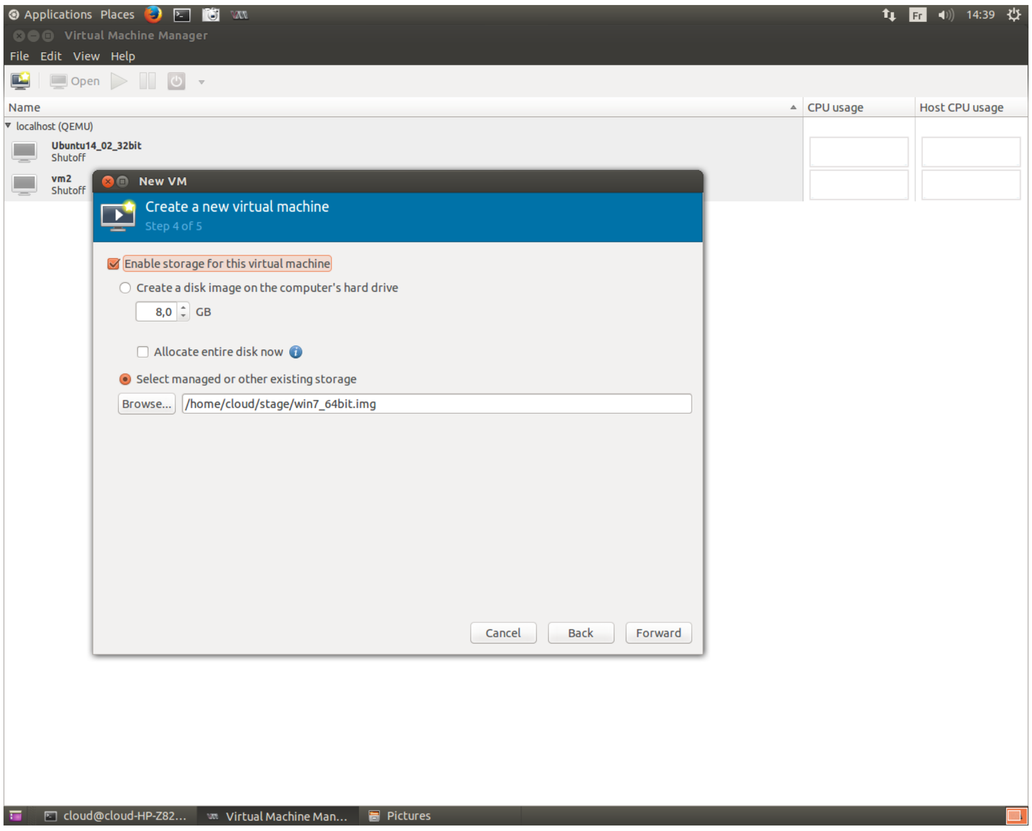
Format: File/Partition format of the volume

Capacity: Maximum size of the volume.

Allocation: Actual size allocated to volume at this time.

Cancel Finish

FIGURE 5.10: Step 3-Create a virtual disk of 20G



Create a new virtual machine

Step 4 of 5

☑ Enable storage for this virtual machine

○ Create a disk image on the computer's hard drive

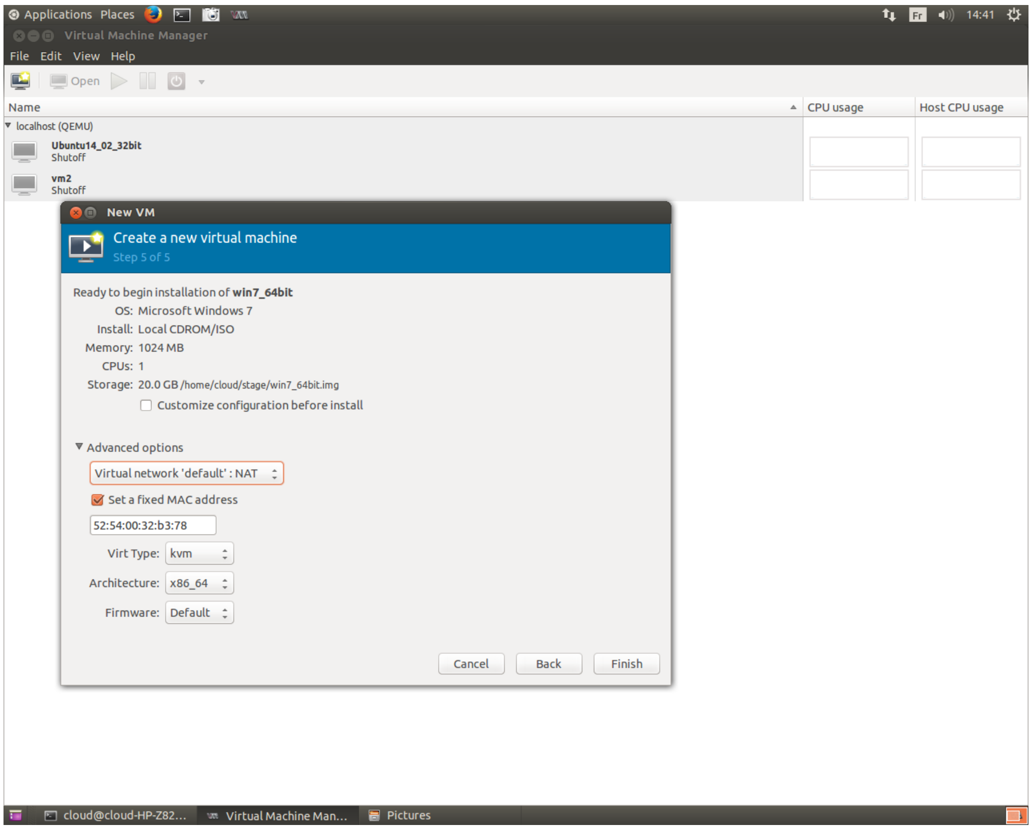
8,0 GB

☐ Allocate entire disk now

● Select managed or other existing storage

Browse... /home/cloud/stage/win7_64bit.img

FIGURE 5.11: Step 4-Indicate virtual disk location for guest installation



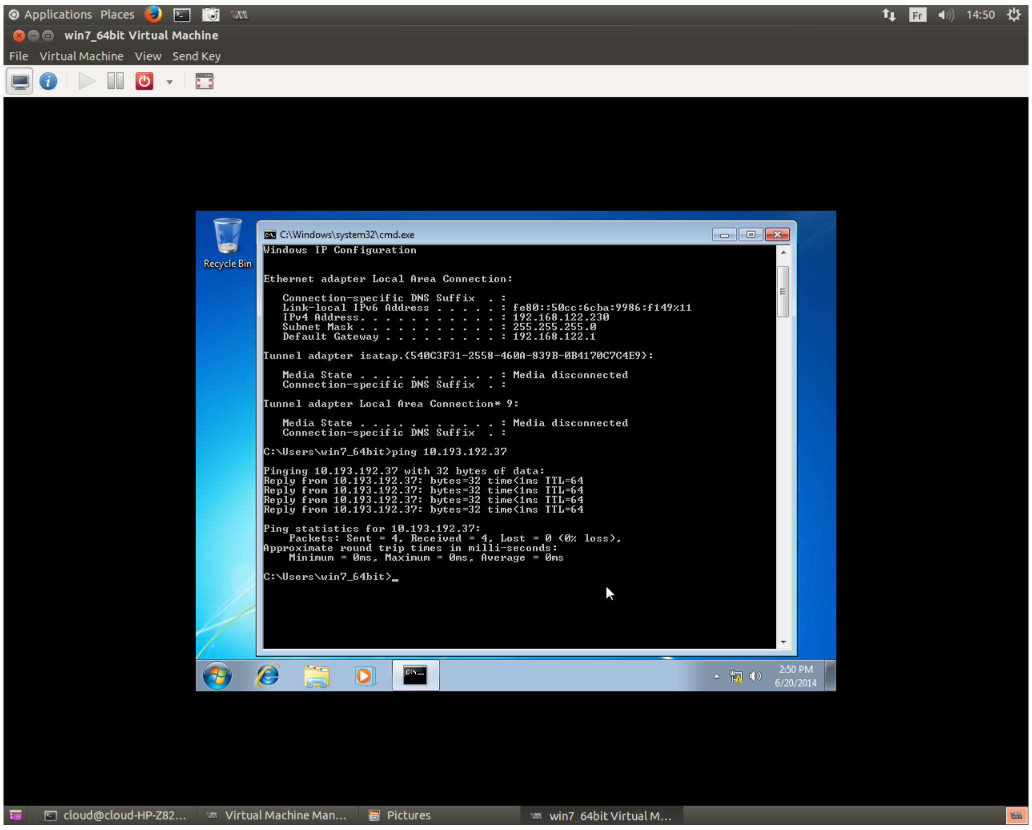
Ready to begin installation of **win7_64bit**
OS: Microsoft Windows 7
Install: Local CDROM/ISO
Memory: 1024 MB
CPUs: 1
Storage: 20.0 GB /home/cloud/stage/win7_64bit.img
☐ Customize configuration before install

Advanced options
Virtual network 'default' : NAT
☑ Set a fixed MAC address
52:54:00:32:b3:78
Virt Type: kvm
Architecture: x86_64
Firmware: Default

FIGURE 5.12: Step 5-Guest installation configuration resume

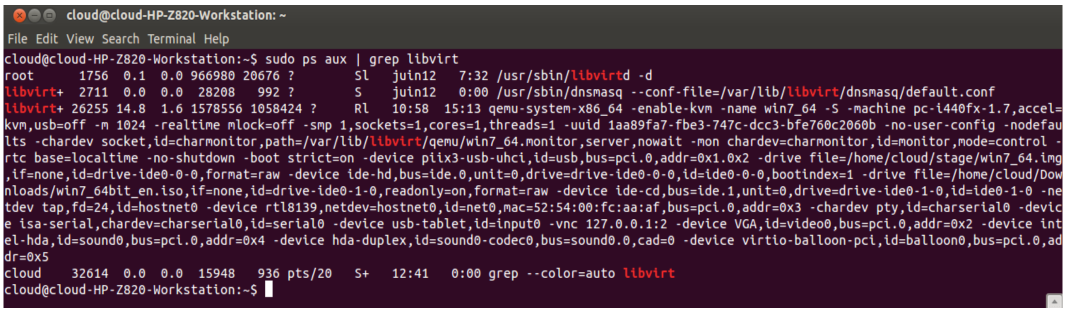
When the installation process is finished, we check guest's IP configuration and ping the host. The result is shown in picture 5.13.

Then verify that the guest machine is launched as a seperate process under Ubuntu by qemu in host machine. The result is shown in Figure 5.14



```
Windows IP Configuration


Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::50cc:6cba:9986:f149%11
   IPv4 Address. . . . . . . . . . . : 192.168.122.230
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.122.1

Tunnel adapter isatap.{540C3F31-2558-460A-839B-0B4170C7C4E9}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter Local Area Connection* 9:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

C:\Users\win7_64bit>ping 10.193.192.37

Pinging 10.193.192.37 with 32 bytes of data:
Reply from 10.193.192.37: bytes=32 time<1ms TTL=64
Reply from 10.193.192.37: bytes=32 time<1ms TTL=64
Reply from 10.193.192.37: bytes=32 time<1ms TTL=64
Reply from 10.193.192.37: bytes=32 time<1ms TTL=64

Ping statistics for 10.193.192.37:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\win7_64bit>_
```

FIGURE 5.13: Guest machine ping host machine


cloud@cloud-HP-Z820-Workstation: ~
File Edit View Search Terminal Help

```
cloud@cloud-HP-Z820-Workstation:~$ sudo ps aux | grep libvirt
root      1756  0.1  0.0 966980 20676 ?        Sl   juin12   7:32 /usr/sbin/libvirtd -d
libvirt+  2711  0.0  0.0  28208   992 ?        S    juin12   0:00 /usr/sbin/dnsmasq --conf-file=/var/lib/libvirt/dnsmasq/default.conf
libvirt+ 26255 14.8  1.6 1578556 1058424 ?     Rl   10:58  15:13 qemu-system-x86_64 -enable-kvm -name win7_64 -S -machine pc-i440fx-1.7,accel=
kvm,usb=off -m 1024 -realtime mlock=off -smp 1,sockets=1,cores=1,threads=1 -uuid 1aa89fa7-fbe3-747c-dcc3-bfe760c2060b -no-user-config -nodefau
lts -chardev socket,id=charmonitor,path=/var/lib/libvirt/qemu/win7_64.monitor,server,nowait -mon chardev=charmonitor,id=monitor,mode=control -
rtc base=localtime -no-shutdown -boot strict=on -device piix3-usb-uhci,id=usb,bus=pci.0,addr=0x1.0x2 -drive file=/home/cloud/stage/win7_64.img
,if=none,id=drive-ide0-0-0,format=raw -device ide-hd,bus=ide.0,unit=0,drive=drive-ide0-0-0,id=ide0-0-0,bootindex=1 -drive file=/home/cloud/Dow
nloads/win7_64bit_en.iso,if=none,id=drive-ide0-1-0,readonly=on,format=raw -device ide-cd,bus=ide.1,unit=0,drive=drive-ide0-1-0,id=ide0-1-0 -ne
tdev tap,fd=24,id=hostnet0 -device rtl8139,netdev=hostnet0,id=net0,mac=52:54:00:fc:aa:af,bus=pci.0,addr=0x3 -chardev pty,id=charserial0 -devic
e isa-serial,chardev=charserial0,id=serial0 -device usb-tablet,id=input0 -vnc 127.0.0.1:2 -device VGA,id=video0,bus=pci.0,addr=0x2 -device int
el-hda,id=sound0,bus=pci.0,addr=0x4 -device hda-duplex,id=sound0-codec0,bus=sound0.0,cad=0 -device virtio-balloon-pci,id=balloon0,bus=pci.0,ad
dr=0x5
cloud    32614  0.0  0.0  15948   936 pts/20   S+   12:41   0:00 grep --color=auto libvirt
cloud@cloud-HP-Z820-Workstation:~$ 
```

FIGURE 5.14: Guest as a qemu process in host machine

Chapter 6

NITRO'S INSTALLATION AND MANIPULATION

6.1 Installation of Nitro

Pfoh has created a website, http://nitro.pfoh.net/setup.html, where he gives a general introduction about Nitro's setup. To reduce the length of this report, it's advised to carefully follow Pfoh's tutorial when installing Nitro. Here we just provide some extra explanations and complements on the basis of his initial tutorial. Remember that Nitro consists of three components:

- Kernel Modules - This component is a fork of KVM. It has been extended to provide additional IOCTL calls that can be leveraged to perform VMI.
- QEMU - This is a fork of QEMU, the component that provides the user land support for KVM. It is only slightly modified such that it exposes direct access to the guest's physical memory, which Nitro can take advantage of.
- Nitro/libnitro - This is the user land component that actually performs VMI. Nitro (this term here represents an executable file/command as opposed to the Nitro project.) calls those APIs defined in libnitro. Remember Nitro is just a prototype implementation which still needs much more enhancements if you want more functionality and could be used as a technique block in other projects. To

facilitate possible further work, we have added plenty of comments in Nitro's source code files. We put Nitro's source files in path /home/cloud/nitro.

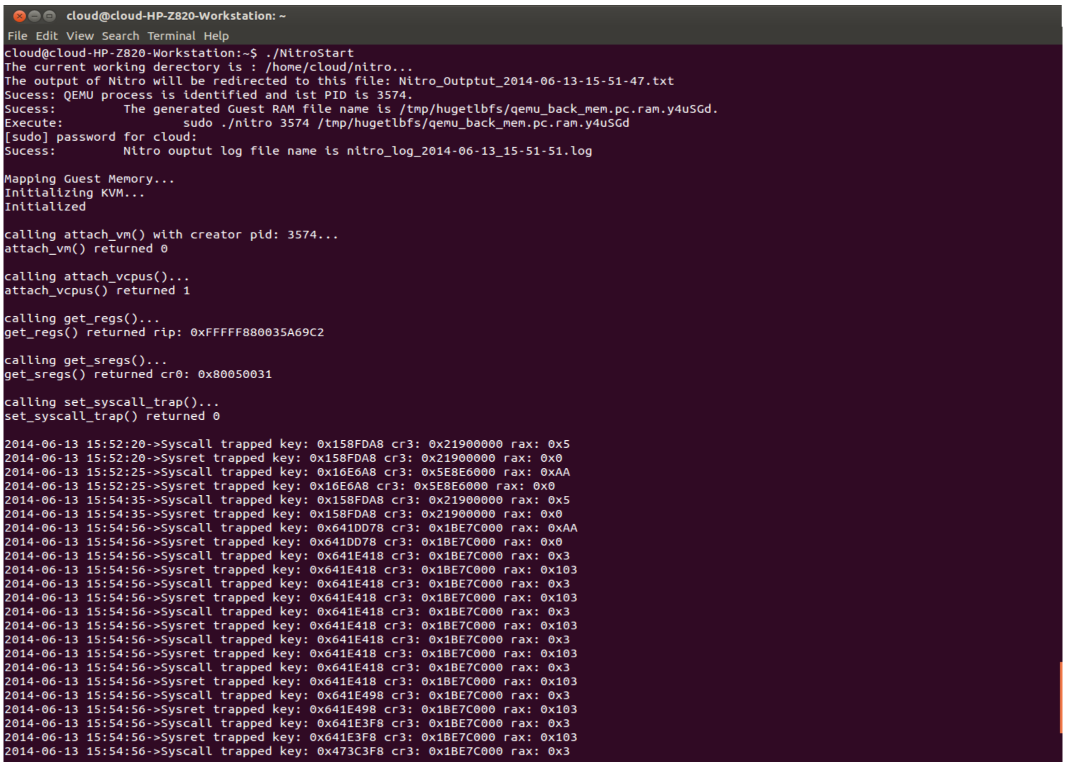
6.2 Manipulation of Nitro

To use Nitro, some preliminary work is necessary, including replace initial KVM modules by a modified one for Nitro, mount a huge pages type device under /tmp directory and finally increase the number of huge pages available to your system. If you are not familiar with huge pages, this tutorial is a good start point: `https://wiki.debian.org/Hugepages`. To automate this work, we have written a bash shell script named "load_mods", which is located in the root of working directory (/home/cloud/load_mods). If not precise, all script files for Nitro are in the same location.

Once the huge table file system is set up, QEMU could be launched. Note that Nitro uses a modified version of QEMU to directly access the guest's physical memory. We put this QEMU in the root of user cloud's working directory (/home/cloud/qemu). When staring the modified QEMU, it works in the same manner as starting vanilla QEMU with KVM support, but you must include option "-mem-path [PATH] -mem-prealloc" in the end of QEMU command line. It is recommended to use a bash script to automate this work. For example, to start guest with WIN7 OS, we write a script named "win7start".

Finally, it's time to launch Nitro to monitor the already running VM. We have also a script "NitroStart" for this work. Note that Nitro command syntax is "./Nitro PID GUEST_RAM_FILE". Figure 6.1 shows a part of output of Nitro.

All the output of Nitro could be devised into two parties. The first part logs the Nitro start process and prints return values for each function invocation. For example, function attach_vcpu( ) returns 1 if Nitro manages to attach to a virtual CPU in guest machine. (In fact, Nitro now support uniquely attachment to one virtual CPU even if the guest has been allocated more than one virtual CPU.). When the invocation of set_syscall_trap() returns zero, Nitro is successfully started and ready to capture system call events. The second part of this output consists of all trapped system call events. Column "Syscall trapped key" means the physical address of related system call service routine. Column "cr3" save the CR3 register's value at the moment of system call trap. And column "rax" indicates the system call code for windows 64 bits operating system. For example,


cloud@cloud-HP-Z820-Workstation: ~
File Edit View Search Terminal Help

```
cloud@cloud-HP-Z820-Workstation:~$ ./NitroStart
The current working derectory is : /home/cloud/nitro...
The output of Nitro will be redirected to this file: Nitro_Outptut_2014-06-13-15-51-47.txt
Sucess: QEMU process is identified and ist PID is 3574.
Sucess:         The generated Guest RAM file name is /tmp/hugetlbfs/qemu_back_mem.pc.ram.y4uSGd.
Execute:                sudo ./nitro 3574 /tmp/hugetlbfs/qemu_back_mem.pc.ram.y4uSGd
[sudo] password for cloud:
Sucess:         Nitro ouptut log file name is nitro_log_2014-06-13_15-51-51.log

Mapping Guest Memory...
Initializing KVM...
Initialized

calling attach_vm() with creator pid: 3574...
attach_vm() returned 0

calling attach_vcpus()...
attach_vcpus() returned 1

calling get_regs()...
get_regs() returned rip: 0xFFFFF880035A69C2

calling get_sregs()...
get_sregs() returned cr0: 0x80050031

calling set_syscall_trap()...
set_syscall_trap() returned 0

2014-06-13 15:52:20->Syscall trapped key: 0x158FDA8 cr3: 0x21900000 rax: 0x5
2014-06-13 15:52:20->Sysret trapped key: 0x158FDA8 cr3: 0x21900000 rax: 0x0
2014-06-13 15:52:25->Syscall trapped key: 0x16E6A8 cr3: 0x5E8E6000 rax: 0xAA
2014-06-13 15:52:25->Sysret trapped key: 0x16E6A8 cr3: 0x5E8E6000 rax: 0x0
2014-06-13 15:54:35->Syscall trapped key: 0x158FDA8 cr3: 0x21900000 rax: 0x5
2014-06-13 15:54:35->Sysret trapped key: 0x158FDA8 cr3: 0x21900000 rax: 0x0
2014-06-13 15:54:56->Syscall trapped key: 0x641DD78 cr3: 0x1BE7C000 rax: 0xAA
2014-06-13 15:54:56->Sysret trapped key: 0x641DD78 cr3: 0x1BE7C000 rax: 0x0
2014-06-13 15:54:56->Syscall trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E418 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E498 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E498 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x641E3F8 cr3: 0x1BE7C000 rax: 0x3
2014-06-13 15:54:56->Sysret trapped key: 0x641E3F8 cr3: 0x1BE7C000 rax: 0x103
2014-06-13 15:54:56->Syscall trapped key: 0x473C3F8 cr3: 0x1BE7C000 rax: 0x3
```

FIGURE 6.1: Truncated output of Nitro for a Win7 64bit VM

a value “AA” in RAX means a system call “NtCreateUserProcess” in windows 7. In fact, in the prototype implementation provided by Pfoh, the type of system call desired is hard coded in file “nitro_main.c”. Nitro captures only those system calls that users require. For more information about windows 64 bits system call table, please refer to this link: http://j00ru.vexillium.org/ntapi_64/.

6.3 Assessment of Nitro

As far as I know, Nitro is actually the first and only VMI application in derivative manner under KVM virtualization platform. Though VMI in derivative mode is perfectly against circumstance technique, the amount of information obtained is rather limited, due to the fact that derivation method principally just concerns and observes hardware-related running state (such as CR3 register, etc.) [34]. Nitro, as a typical implementation under the guide of derivation method, is no different. Currently, Nitro is just able to access vCPU registers and trap system calls of type “syscall/sysret” (In fact, system call is possible to be invoked in other manners such as assembly instruction int 0x80.), even though Pfoh promises to enrich Nitro’s functionality list in his spare time.

Just as Pfoh has stated in his project website, Nitro at present has a relatively limited suit of functionalities compared with those VMI tools like LibVMI in out-of-band. Concretely, it presents the following limitations:

- No documentation about use of Nitro, which is really painful for beginners
- No support for AMD CPUs
- No support for multi-core guests
- Only 64-bit windows guests are supported for now
- No cooperation with other tools like virsh/virt-manager

The final conclusion for Nitro is that, as a prototype implementation for VMI derivative pattern theory, it is a good start point to study, manipulate and modify for the purpose of study, For example, to anatomy its modification about KVM modules, we could learn how to enhance KVM virtualization support. As a development framework, it is not enough mature to simplify development work, due to the fact Nitro itself is still a project in progress.

The above conclusion could be extended to a conclusion for derivative pattern for VMI application. Inferring information from hardware architecture to mitigate semantic gap, this pattern ignores OS-related semantic knowledge to get a better portability feature. However, to get high-level information, to implement monitoring task for example, OS-related semantic knowledge is somewhat indispensable. We suggest that:

- Derivative pattern could be used alongside with Out-of-band pattern to provide a complementary view for guest running state
- Derivative is suitable to be used as the last defense line for security purpose

6.4 Potentialities of CR3 in VMI

By reading and analyze carefully those VMI applications leveraging CR3 register, we could get some conclusions:

Firstly, the functionality of CR3 register, according to Intel's specification [10] and Pfoh's exploiting [34], is just used to hold the address of the top-level page directory for the currently executing process. This value could be used to uniquely represent a process. No others special functionality. Secondly, the prior work about deriving or inferring guest information from CR3 register is all about countering security threats such as identifying hidden processes and stopping them in guest, while our study about CR3 register is to investigate if CR3 register could provide some information about monitoring network traffic. Accordingly, by nature, countering security threats needs less information at the level of process than monitoring task. For countering a malicious process, we are allowed to identify the process address space and delete this allocated address space to protect guest and it is not necessary to know the PID or process name or application name related to the killed dangerous process. However, in the context of monitoring task, for example, monitoring network traffic on the basis of process, in my view, it needs much more detailed information.

Chapter 7

LibVMI&Volatility MANIPULATION

7.1 INTRODCTION

Recall that the objective of this internship derives from a predication made by Pfoh in his doctoral thesis in chapter 6 Derivative Method [16]: These methods (Derivative method) allow one to track network connections on a per-process basis in a completely guest operating system agnostic manner.

Frankly, a Virtual Machine Introspection application VMWall [7] has implemented almost the similar functionality. However, VMWall's implementation is based on delivery pattern, thus it has a narrow set of operating system support and requires some preliminary configuration files to work. All above characteristics hinder wide application of this kind of VMI application. Therefore, what Pfoh has declared actually attract my attention is the capacity presented by his derivative method to achieve the same functionality but in a "completely guest operating system agnostic manner".

To implement his claim, this task could be naturally treated from two plans: network plan and process plan. With regard to network plan, Pfoh gives the theoretical groundwork about his predication: All guests I/O activities, including network traffic, have to rely on the hypervisor. In other words, performing such network traffic monitoring is a matter of tapping into the virtual I/O device within the hypervisor and interpreting the data. Thus, it is not difficult to get to know network-related information such as the source/destination IP address, source/destination port number, for each network connection. Then we consider the identification of processes in a running guest. According to him, all running processes could be identified and represented uniquely by its corresponding CR3 register value. Assuming that network plan for implementation of his claim is finished and the running process list is obtained, it remains how to find and associate the corresponding process information (represented by CR3 value) according to obtained network information (represented by source/destination IP address and port). For VMWall, the key attribute to do this association is connection port number. For a derivative method, still back to Pfoh's work, he has stated that one could identify a unique process by inspecting CR3 register. Unfortunately, he didn't expatiate how to associate CR3 register value and network information. We don't know which attribute is shared by network connection and CR3 register. In fact, this is the most important part in the course of implementing Pfoh's predication.

Stuck with this step for longtime, in the meantime, I have tried to explore if the system call (such as read(), wirte(), socket()) related to network I/O, or I/O interrupt could associate network connection and a CR3-represented process. Due to the lack of documentation in this domain, I have none significate discovery. Hence I tried to think in our objective in another perspective. Firstly, the reason why we prefer derivative method is its OS-agnostic property, if some powerful VMI tools of out-of-band method could provide the same or similar OS-agnostic property, we could give it a try. After all, out-of-band VMI research domain is much more active than derivative method and therefore provides more powerful tools. Secondly, our exploration about derivative method is to watch its possible contribution to monitoring task. In this angle, derivative method is inherently limited compared to delivery method, because derivative method works in low-level and could not get high-level information (Obviously, high-level information is usually related to operating system kernel data structure). Thirdly, we could try to firstly implement a network connection monitor in out-of-band manner and may get some inspiration for a derivative method implementation. In addition, this delivery method implementation could be used as a reference for future derivative method implementation.

During the searching online, I have noticed that currently leveraging forensic memory analysis tools, also known as FMA, is a popular tendency in Virtual Machine Introspection application development [35]. The remains of this section is aimed to record the exploration of how to leverage FMA tool such as Volatility with VMI tools such as LibVMI for running virtual machine.

7.2 FORENSIC MEMORY ANALYSIS TOOL

Forensic Memory Analysis is the science of using a memory image to determine information about running programs, the operating system , and the overall state of a computer [36]. Because the analysis is highly dependent on the operating system, it has been divided into the following categories:

- Linux Memory Analysis
- Mac OS X Memory Analysis
- Windows Memory Analysis

Similar with Virtual Machine Introspection, the FMA community is also faced with the semantic gap problem to extract forensically relevant information from dumps of physical memory. The only difference between VMI and FMA regarding semantic gap lies in that VMI is used to monitor a running guest's memory (dynamic file) while FMA initially is used to analyze memory dump file (static file). Figure 7.1 shows the relationship between FMA and VMI.

Up to now, FMA community has made plenty achievements to mitigate semantic gap. Many excellent FMA tools, such as Volatility framework, are available. The following figure shows the comparison between the famous VMI tool LibVMI and Volatility. Since after many years developments, Volatility has a wide set of memory analysis support, from Android to Mac, from Linux to Windows. Hence, leveraging this characteristic, it is possible to create an OS-agnostic VMI application, if we have some approaches to make Volatility treat virtual machine's memory as a recognizable file. Fortunately, LibVMI, a VMI framework based on out-of-band method, has developed a python-wrapper which allows Volatility functioning for a running guest.

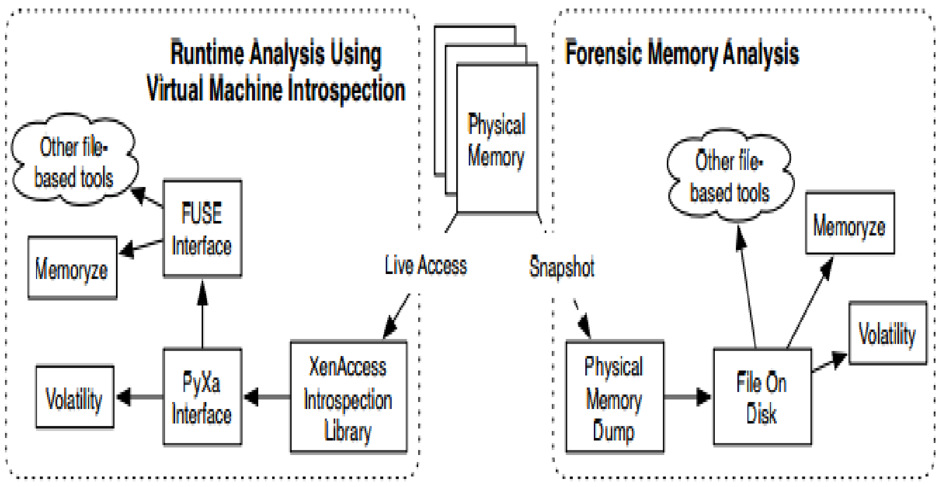


FIGURE 7.1: Comparison of VMI and FMA [14]

7.3 INSTALLATION LIBVMI AND VOLATILITY IN KVM

This section will introduce how to install LibVMI&Volatility in Ubuntu14.04, where KVM virtualization platform has been established with success. With regard to how to establish KVM platform in Ubuntu14.04, this link http://adywp.blogs.unhas.ac.id/2014/05/installing-kvm-on-ubuntu-14-04/ is strongly recommended.

It is worth to mentioning that the libvirt library, installed by Ubuntu's package manager (for example by apt-get), does not support QMP command, which is necessary for QEMU emulator to access directly target guest's physical memory. Similarly, QEMU utility installed by default does not provide functions to access virtual machine's memory. Hence, to use LibVMI in KVM platform, the first step is to get respectively source codes for libvirt and QEMU, modify QEMU to add VM physical memory access code and install these utilities from source code.

To get libvirt source code, open a terminal and change into your destination directory (in our case, we plan to arrange all source codes required under a directory called "libvmi") and issue this command in a terminal:

```
#  mkdir  /libvmi

# cd  /libvmi

# wget http://libvirt.org/sources/libvirt-1.2.6.tar.gz

# tar zxvf libvirt-1.2.6.tar.gz
# cd libvirt-1.2.6
```

When compiling and installing libvirt from source code, notice that an older version (version 1.2.2) has been already installed under path /usr/bin, to avoid any possible confusion, we need to update libvirt by overriding the old one:

```
#  ./autogen.sh
# ./configure --prefix=/usr --localstatedir=/var --sysconfdir=/etc
# make
# sudo make install
# sudo ldconfig
```

Then, do not forget to modify libvirt configuration file and restart libvrit daemon. Modify /etc/libvirt/libvirtd.conf, uncomment this line:

```
# auth_unix_rw = \none"
```

To restart libvirtd, type the following commands:

```
#  sudo /etc/init.d/libvirt-bin stop
# sudo /etc/init.d/libvirt-bin start
```

In term of QEMU, we need to firstly know why and how to modify its source code. What the patch actually does is to use Qemu Machine Protocol (QMP) mechanism for LibVMI to pass the GPA (Guest Physical Address), and call the internal function "cpu_physical_memory_map" located in qemu/exec.c in Qemu, and finally get the mapped HVA (Host Virtual Address) back. More details about how to write a patch for a certain version of QEMU are available in this link: http://ytliu.info/blog/2014/03/27/kvm-support-in-libvmi. Given that a QEMU patch for QEMU 1.6 is provided, we use git to clone branch 1.6 version of QEMU:

```
#  mkdir  /libvmi/qemu-stable-1.6
#cd qemu-stable-1.6
#git init
#git remote add -t state-1.6 -f origin https://github.com/qemu/qemu.git
#git checkout stable-1.6
```

Supposing that the path of patch file for QEMU 1.6 is /libvmi/kvm-1.6-patch.patch, to patch QEMU, issue this command:

```
#  patch -p1 < ../kvm-1.6-patch.patch
# ./configure --prefix=/usr --target-list="386-softmmu x86_64-softmmu"
```

By default (without –target-list option), 'configure' will prepare many machine type emulator, and it takes long time to compile.

```
#  make
# sudo make install
```

In addition, to assure the functioning of LibVMI, all the following additional packages are required to be installed:

```
#  sudo apt-get install zlib1g-dev libglib2.0-dev libpixman-1-dev libfdt-dev libtool libsdl1.2-dev
# sudo apt-get install libbison-dev flex libyajl-dev check autopoint python-dev libxslt1-dev
xsltproc
# sudo apt-get install libdevmapper-dev libpciaccess-dev libnl-dev w3c-dtd-xhtml libjansson-dev
libfuse-dev
```

Now all prerequisites works are done to install LibVMI. To get its source code, type this command:

```
#  cd  /libvmi
# git clone git://github.com/bdpayne/libvmi.git
```

Then change into LibVMI directory

```
#  cd libvmi # ./autogen.sh
# ./configure
# make
# sudo make install
# sudo ldconfig
```

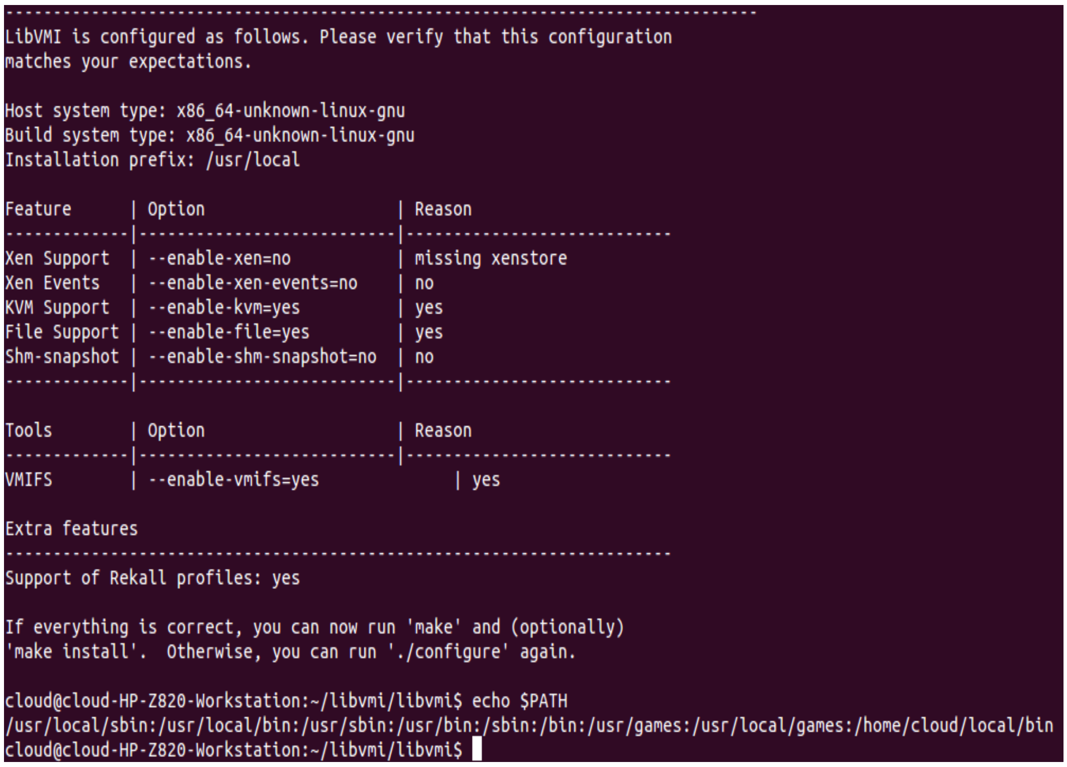
If all required packages are present in system, after running. /configure command, we should the output shown in Figure 7.2.

LibVMI will be installed in path /usr/local/bin. It is worth to mentioning that after update of QEMU and libvirt, some KVM's virtual machines, which are installed under the help of older libvirt version, will encounter some problems to start. This is caused by the name convention difference in virtual machine XML configuration file.

After modifying target guest virtual machine's configuration, reload it to take effect

```
# sudo service livirt-vin reload
```

Then all guests who have problems to start now could work normally as before. To verify the correct installation of LibVMI, we need to execute the example "process-list" shipped with LibVMI. This example needs some configuration file to work. The details on this file can be read here: https://code.google.com/p/vmitools/wiki/LibVMIInstallation. In our situation, we put this configuration "libvmi.conf" under path /etc/. The most difficult task to complete the LibVMI configuration file is how to



```
-----------------------------------------------------------------------
LibVMI is configured as follows. Please verify that this configuration
matches your expectations.

Host system type: x86_64-unknown-linux-gnu
Build system type: x86_64-unknown-linux-gnu
Installation prefix: /usr/local

Feature      | Option                   | Reason
-------------|--------------------------|--------------------------
Xen Support  | --enable-xen=no          | missing xenstore
Xen Events   | --enable-xen-events=no   | no
KVM Support  | --enable-kvm=yes         | yes
File Support | --enable-file=yes        | yes
Shm-snapshot | --enable-shm-snapshot=no | no
-------------|--------------------------|--------------------------

Tools        | Option                   | Reason
-------------|--------------------------|--------------------------
VMIFS        | --enable-vmifs=yes             | yes

Extra features
----------------------------------------------------------------
Support of Rekall profiles: yes

If everything is correct, you can now run 'make' and (optionally)
'make install'.  Otherwise, you can run './configure' again.

cloud@cloud-HP-Z820-Workstation:~/libvmi/libvmi$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/home/cloud/local/bin
cloud@cloud-HP-Z820-Workstation:~/libvmi/libvmi$
```

Figure 7.2: Output of ./configure for LibVMI

get "offset" value for certain kernel data structure. Fortunately, LibVMI has provided specific tools for this task.

Ignoring the process of how to edit this file, the following is the part of libvmi.conf:

```
vm2 {
ostype = "Linux";
sysmap = "/home/cloud/etc/System.map-2.6.32-38-generic";
linux_name = 0x30c;
linux_tasks = 0x1d8;
linux_mm = 0x1f4;
linux_pid = 0x214;
linux_pgd = 0x28;
}

vm1{
ostype = "Windows";
win_tasks = 0x188;
win_pdbase = 0x28;
```

win_pid = 0x180;

win_pname = 0x2e0;

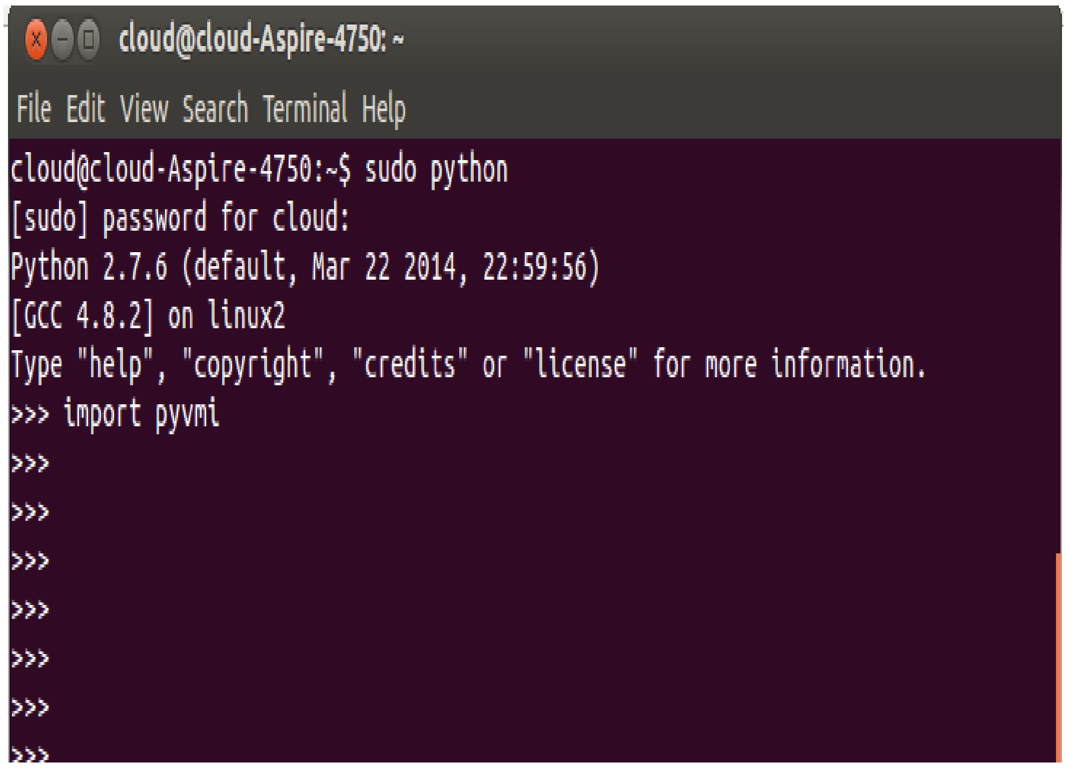
}

Executing the following command, we should see the list of currently running processes inside vm2.

```
# sudo process-list vm2
```

From now on, we could say that LibVMI has been successfully installed in our KVM virtualization platform. LibVMI allows cooperating with Volatility Forensic Memory Analysis framework by providing a python wrapper; due to the fact that Volatility is written in Python while LibVMI in C. To install this Python wrapper, change into the folder "pyvmi" and issue commands:

```
# python setup.py build

# sudo python setup.py install
```


cloud@cloud-Aspire-4750: ~
File Edit View Search Terminal Help

```
cloud@cloud-Aspire-4750:~$ sudo python
[sudo] password for cloud:
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyvmi
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

Figure 7.3: Import Pyvmi in Python interactive Shell

Run python in a terminal and try to import pyvmi module, nothing output means that LibVMI has been encapsulated as module into Python. Now we could invoke LibVMI API in Python script.

Now it's time to get Volatility.

```
# cd  /libvmi


# wget https://volatility.googlecode.com/files/volatility-2.3.tar.gz


# tar zxvf volatility-2.3.tar.gz


# cd volatility-2.3.tar.gz
```

In term of installing Volatility's code, we have two choices, each has its own advantages and disadvantages.

1) Extract the archive and run setup.py. This way is useful when we want to import Volatility as module in Python script, however, it is not convenient for upgrading or uninstalling.

2) Extract the archive to a directory of your choice. When you want to use Volatility just do python /path/to/directory/vol.py. This is a cleaner method since no files are ever moved outside of your chosen directory, which is convenient for possible update of Volatility in the future. The cost is that Volatility could not be used as a library in Python script.

For our case, we choose to run setup.py for the purpose of using Volatility as a module. Now we manage to install LibVMI and Volatility in our KVM experimentation platform. The following section will talk about my manipulation.

7.4 Explore Volatility with LibVMI

This section is used to record the exploration course of exploring the usage of Volatility and LibVMI for running KVM virtual machines. The first section is about a general introduction about Volatility.

7.4.1 Introduction about Volatility

The Volatility Framework is a completely open-source collection of tools, implemented in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples. The extraction techniques are performed completely independent of the system being investigated but offer visibility into the runtime state of the system. The framework is intended to introduce people to the techniques and complexities associated with extracting digital artifacts from volatile memory samples and provide a platform for further work into this exciting area of research. The official documentation is very complete and is available here : `http://code.google.com/p/volatility/wiki/VolatilityIntroduction?tm=6`.

7.4.2 Volatility Usage

Since the installation is introduced previously, here we talk about directly the usage of this powerful tool. Briefly, the most basic volatility commands are constructed as shown below:

```
# python path/to/vol.py [plugin] -f [image] --profile=[profile]
```

Placeholder [plugin] in above command line represents the functionality provided by Volatility. For example "pslist" plugin allows listing all currently running processes. [image] means the path to the target memory dump. It could be in various types depending supported address space, such as raw dd style format or LiME [37] format for Linux. [profile] indicates which memory layout or kernel data structure will be used to investigate input memory sample file. One reason why Volatility is so useful and popular is due to complete and predefined profile for Windows.

Replace plugin with the name of the plugin to use (pslist, netscan, linxu_netstat,etc.), image with the file path to your memory image, and profile with the name of the profile (such as Win7SP1x64, the default profile is always WinXPSP3x86).

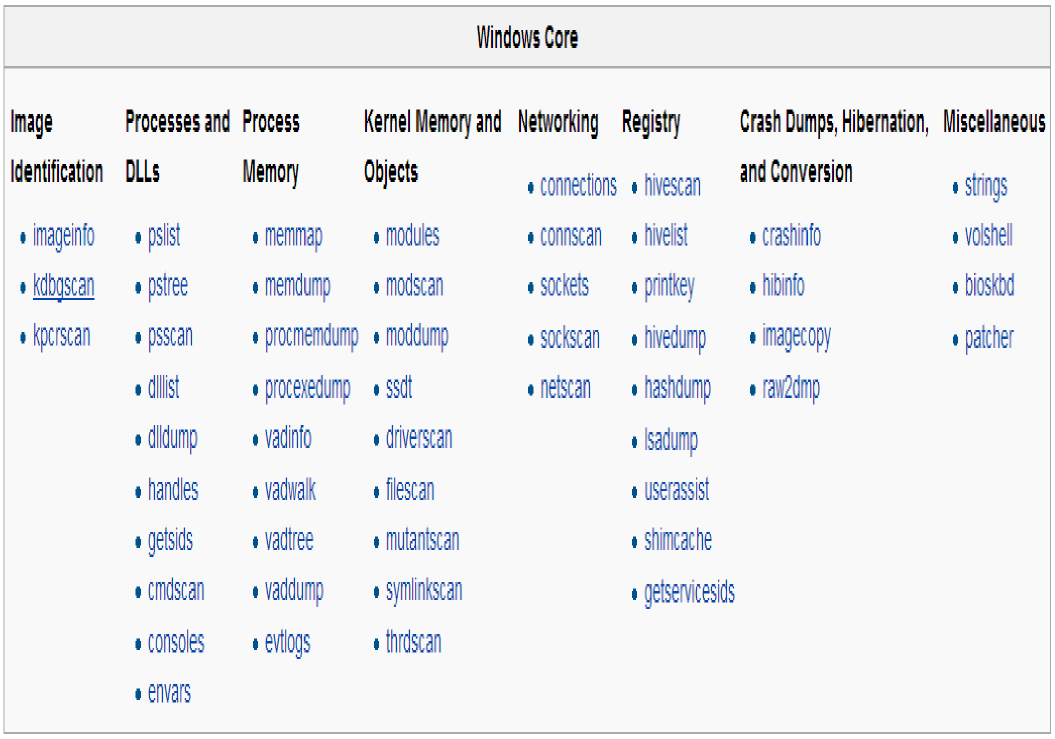
For example, imaging a Windows guest memory dump named "win7.dd" is at our disposal. Now we want to investigate which network connections are established. The following command is used to achieve this:

```
# sudo python vol.py netscan -f /path/to/win7.dd {profile=Win7SP1x86
```

For everything beyond this example, such as controlling the output format, listing the available plugins and profiles, or supplying plugin-specific options, see the rest of the text below. https://code.google.com/p/volatility/wiki/VolatilityUsage23

7.4.3 Plugin

As mentioned above, the functionalities of Volatility are in implemented in form of plugin and could be extended. Initially, Volatility is used uniquely for Windows memory analysis, thus it provides up to now variety of plugins for Windows. This plugins could be grouped into the those categories shown in Figure 7.4:



Windows Core

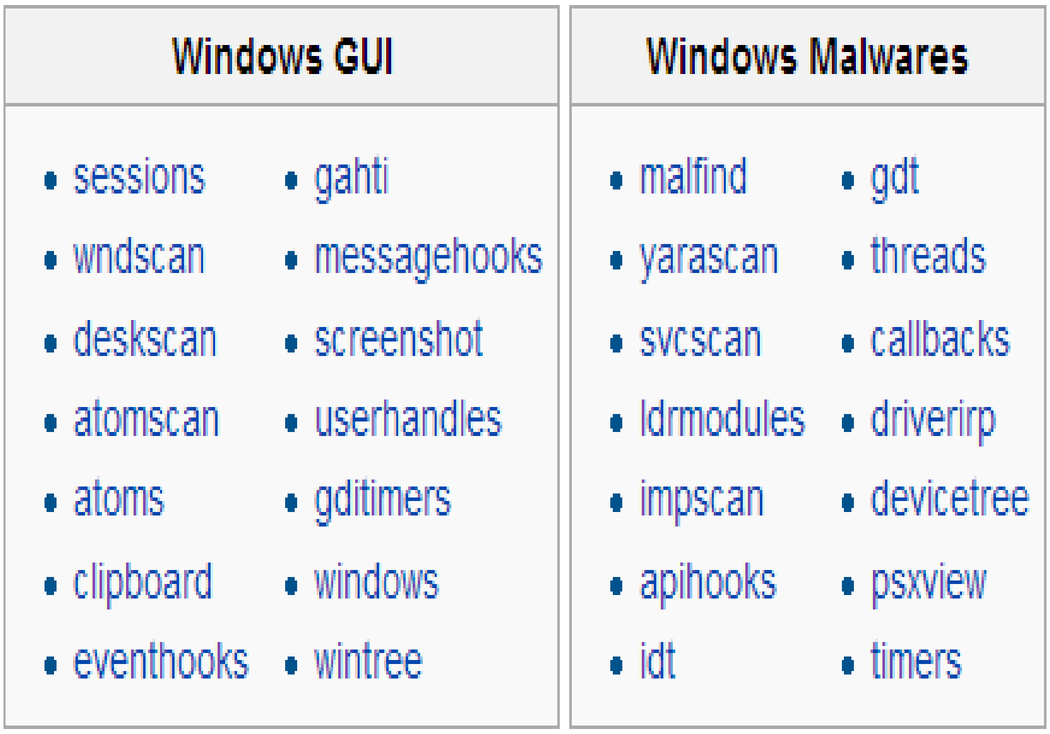
Image Identification	Processes and DLLs	Process Memory	Kernel Memory and Objects	Networking	Registry	Crash Dumps, Hibernation, and Conversion	Miscellaneous
• imageinfo	• pslist	• memmap	• modules	• connections	• hivescan	• crashinfo	• strings
• kdbgscan	• pstree	• memdump	• modscan	• connscan	• hivelist	• hibinfo	• volshell
• kpcrscan	• psscan	• procmemdump	• moddump	• sockets	• printkey	• imagecopy	• bioskbd
	• dlllist	• procexedump	• ssdt	• sockscan	• hivedump	• raw2dmp	• patcher
	• dlldump	• vadinfo	• driverscan	• netscan	• hashdump		
	• handles	• vadwalk	• filescan		• lsadump		
	• getsids	• vadtree	• mutantscan		• userassist		
	• cmdscan	• vaddump	• symlinkscan		• shimcache		
	• consoles	• evtlogs	• thrdscan		• getservicesids		
	• envars						

FIGURE 7.4: Windows Core Plugin List [4]

Thus with Volatility, we could get a rather clear visibility for the security or monitoring purpose, for the running state of the target machine if we have its memory sample file.

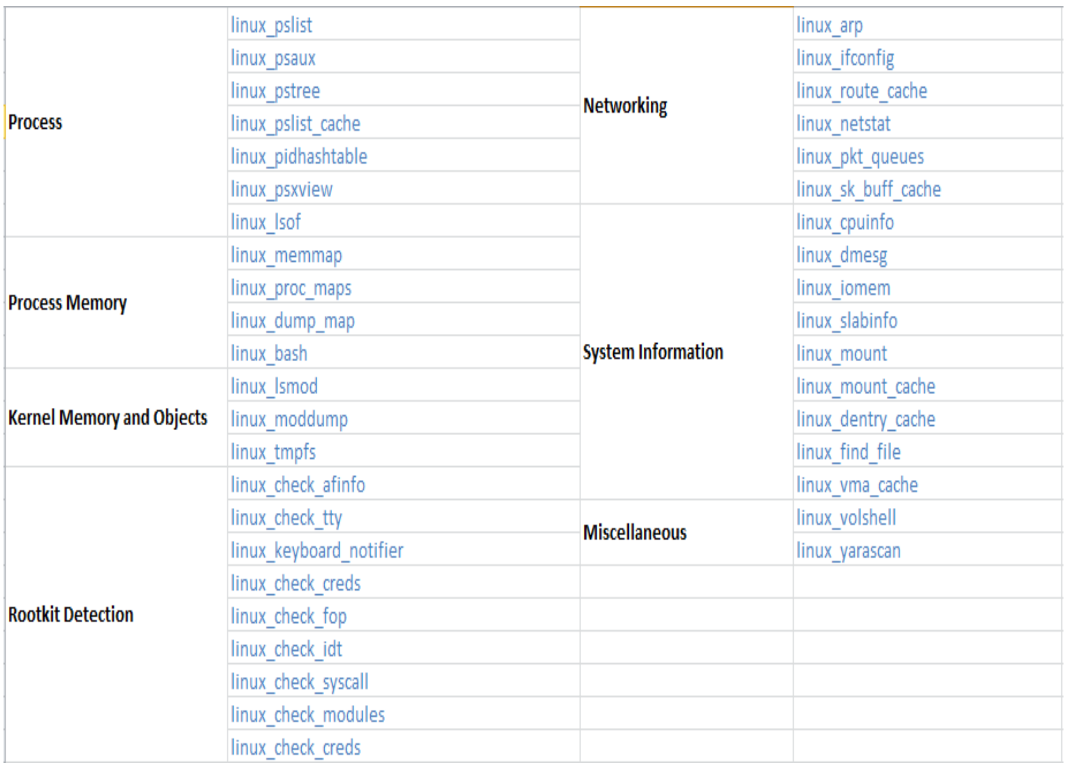
Before Volatility version 2.1, Volatility does not support memory forensic analysis for Linux. At the moment, another FMA tool called Volitilinux is used for this task. Volitilinux could be regarded as counterpart of Volatility for Linux. From Volatility version 2.2, Volatility has incorporated Volitlinux's functionality and has a more wide range OS support.

In term of Linux, available plugins are resumed in Figure 7.6.



Windows GUI		Windows Malwares	
sessions	gahti	malfind	gdt
wndscan	messagehooks	yarascan	threads
deskscan	screenshot	svcscan	callbacks
atomscan	userhandles	ldrmodules	driverirp
atoms	gditimers	impscan	devicetree
clipboard	windows	apihooks	psxview
eventhooks	wintree	idt	timers

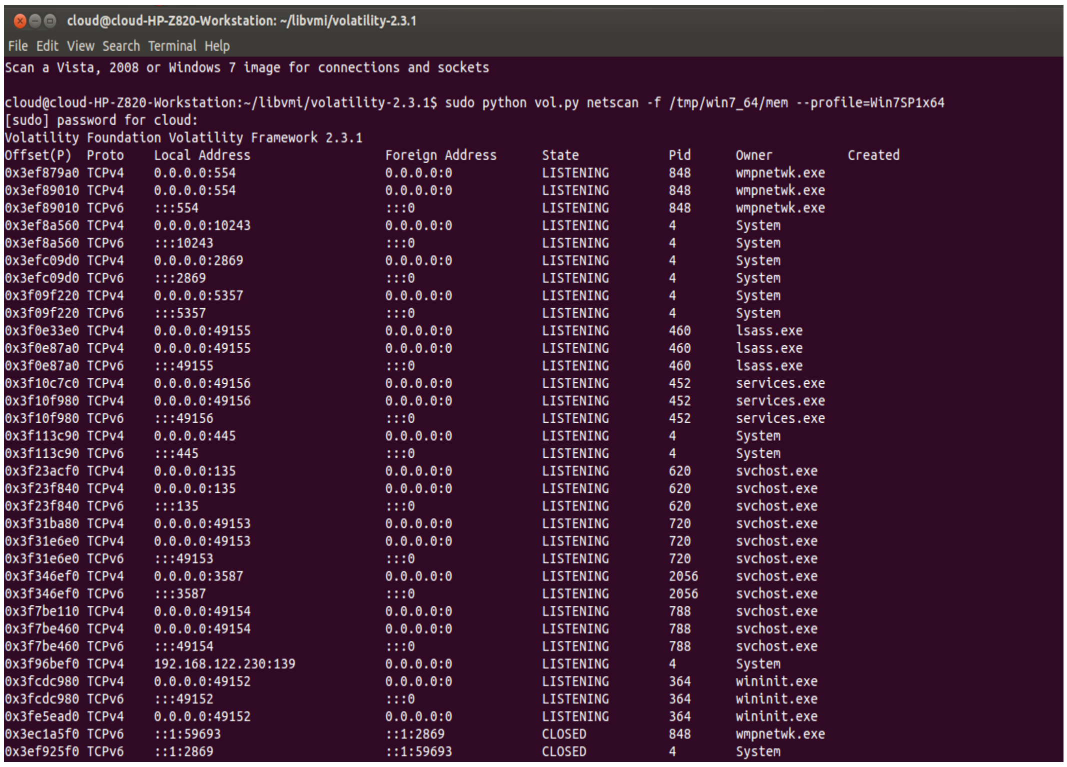
FIGURE 7.5: Windows GUI and Malware Plugins List [4]



Process	linux_pslist	Networking	linux_arp
	linux_psaux		linux_ifconfig
	linux_pstree		linux_route_cache
	linux_pslist_cache		linux_netstat
	linux_pidhashtable		linux_pkt_queues
	linux_psxview		linux_sk_buff_cache
	linux_lsof	System Information	linux_cpuinfo
Process Memory	linux_memmap		linux_dmesg
	linux_proc_maps		linux_iomem
	linux_dump_map		linux_slabinfo
	linux_bash		linux_mount
Kernel Memory and Objects	linux_lsmod		linux_mount_cache
	linux_moddump		linux_dentry_cache
	linux_tmpfs		linux_find_file
Rootkit Detection	linux_check_afinfo		linux_vma_cache
	linux_check_tty	Miscellaneous	linux_volshell
	linux_keyboard_notifier		linux_yarascan
	linux_check_creds		
	linux_check_fop		
	linux_check_idt		
	linux_check_syscall		
	linux_check_modules		
	linux_check_creds		

FIGURE 7.6: Linux Memory Forensic Plugin [27]

In this article, we will not present all the available plugins. Here we pick up some typical and useful command to present the power of Volatility. For example, Figure 7.7 demonstrates that "netscan" command helps us to get all the currently established network connections with their corresponding process.


cloud@cloud-HP-Z820-Workstation: ~/libvmi/volatility-2.3.1
File Edit View Search Terminal Help

```
Scan a Vista, 2008 or Windows 7 image for connections and sockets

cloud@cloud-HP-Z820-Workstation:~/libvmi/volatility-2.3.1$ sudo python vol.py netscan -f /tmp/win7_64/mem --profile=Win7SP1x64
[sudo] password for cloud:
Volatility Foundation Volatility Framework 2.3.1
Offset(P)  Proto    Local Address                  Foreign Address      State            Pid      Owner          Created
0x3ef879a0 TCPv4    0.0.0.0:554                    0.0.0.0:0            LISTENING        848      wmpnetwk.exe
0x3ef89010 TCPv4    0.0.0.0:554                    0.0.0.0:0            LISTENING        848      wmpnetwk.exe
0x3ef89010 TCPv6    :::554                         :::0                 LISTENING        848      wmpnetwk.exe
0x3ef8a560 TCPv4    0.0.0.0:10243                  0.0.0.0:0            LISTENING        4        System
0x3ef8a560 TCPv6    :::10243                       :::0                 LISTENING        4        System
0x3efc09d0 TCPv4    0.0.0.0:2869                   0.0.0.0:0            LISTENING        4        System
0x3efc09d0 TCPv6    :::2869                        :::0                 LISTENING        4        System
0x3f09f220 TCPv4    0.0.0.0:5357                   0.0.0.0:0            LISTENING        4        System
0x3f09f220 TCPv6    :::5357                        :::0                 LISTENING        4        System
0x3f0e33e0 TCPv4    0.0.0.0:49155                  0.0.0.0:0            LISTENING        460      lsass.exe
0x3f0e87a0 TCPv4    0.0.0.0:49155                  0.0.0.0:0            LISTENING        460      lsass.exe
0x3f0e87a0 TCPv6    :::49155                       :::0                 LISTENING        460      lsass.exe
0x3f10c7c0 TCPv4    0.0.0.0:49156                  0.0.0.0:0            LISTENING        452      services.exe
0x3f10f980 TCPv4    0.0.0.0:49156                  0.0.0.0:0            LISTENING        452      services.exe
0x3f10f980 TCPv6    :::49156                       :::0                 LISTENING        452      services.exe
0x3f113c90 TCPv4    0.0.0.0:445                    0.0.0.0:0            LISTENING        4        System
0x3f113c90 TCPv6    :::445                         :::0                 LISTENING        4        System
0x3f23acf0 TCPv4    0.0.0.0:135                    0.0.0.0:0            LISTENING        620      svchost.exe
0x3f23f840 TCPv4    0.0.0.0:135                    0.0.0.0:0            LISTENING        620      svchost.exe
0x3f23f840 TCPv6    :::135                         :::0                 LISTENING        620      svchost.exe
0x3f31ba80 TCPv4    0.0.0.0:49153                  0.0.0.0:0            LISTENING        720      svchost.exe
0x3f31e6e0 TCPv4    0.0.0.0:49153                  0.0.0.0:0            LISTENING        720      svchost.exe
0x3f31e6e0 TCPv6    :::49153                       :::0                 LISTENING        720      svchost.exe
0x3f346ef0 TCPv4    0.0.0.0:3587                   0.0.0.0:0            LISTENING        2056     svchost.exe
0x3f346ef0 TCPv6    :::3587                        :::0                 LISTENING        2056     svchost.exe
0x3f7be110 TCPv4    0.0.0.0:49154                  0.0.0.0:0            LISTENING        788      svchost.exe
0x3f7be460 TCPv4    0.0.0.0:49154                  0.0.0.0:0            LISTENING        788      svchost.exe
0x3f7be460 TCPv6    :::49154                       :::0                 LISTENING        788      svchost.exe
0x3f96bef0 TCPv4    192.168.122.230:139            0.0.0.0:0            LISTENING        4        System
0x3fcdc980 TCPv4    0.0.0.0:49152                  0.0.0.0:0            LISTENING        364      wininit.exe
0x3fcdc980 TCPv6    :::49152                       :::0                 LISTENING        364      wininit.exe
0x3fe5ead0 TCPv4    0.0.0.0:49152                  0.0.0.0:0            LISTENING        364      wininit.exe
0x3ec1a5f0 TCPv6    ::1:59693                      ::1:2869             CLOSED           848      wmpnetwk.exe
0x3ef925f0 TCPv6    ::1:2869                       ::1:59693            CLOSED           4        System
```

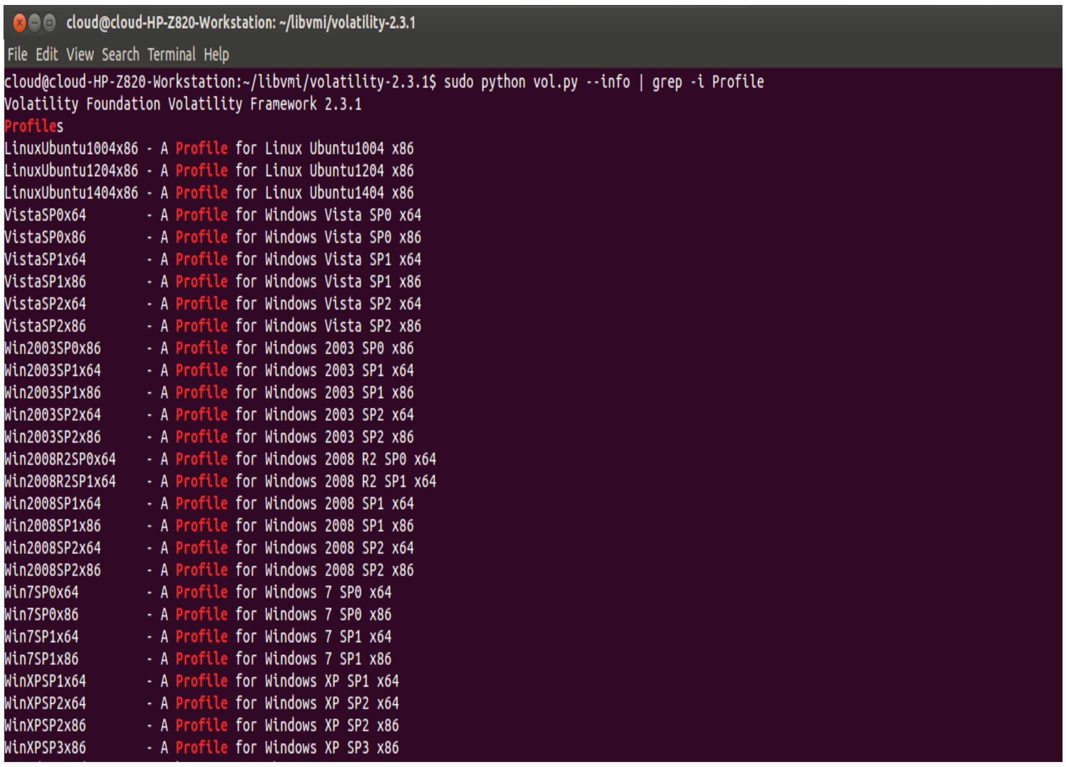
Figure 7.7: Volatility netscan(for windows) plugin's output

Up to now, the actual list of available plugins is long and grows quickly due to the strong development community. We could leverage these plugins to develop our own VMI applications or extend the list of available plugins.

7.4.4 Profile

Volatility is actually an out-of-band method to mitigate the semantic gap, due to the fact that before executing forensic memory, some configuration files about operating system kernel data structures are required. This kind of configuration file is usually called "Profile". The command "sudo python path/to/vol.py –info — grep -i Profile" returns the profile list (Figure 7.8)currently supported by Volatility.

Those who need an attention is, Volatility by default has provided profiles for all existing Windows series OS, because Windows OS's kernel versions are relatively stable.



```
cloud@cloud-HP-Z820-Workstation: ~/libvmi/volatility-2.3.1
File Edit View Search Terminal Help
cloud@cloud-HP-Z820-Workstation:~/libvmi/volatility-2.3.1$ sudo python vol.py --info | grep -i Profile
Volatility Foundation Volatility Framework 2.3.1
Profiles
LinuxUbuntu1004x86 - A Profile for Linux Ubuntu1004 x86
LinuxUbuntu1204x86 - A Profile for Linux Ubuntu1204 x86
LinuxUbuntu1404x86 - A Profile for Linux Ubuntu1404 x86
VistaSP0x64         - A Profile for Windows Vista SP0 x64
VistaSP0x86         - A Profile for Windows Vista SP0 x86
VistaSP1x64         - A Profile for Windows Vista SP1 x64
VistaSP1x86         - A Profile for Windows Vista SP1 x86
VistaSP2x64         - A Profile for Windows Vista SP2 x64
VistaSP2x86         - A Profile for Windows Vista SP2 x86
Win2003SP0x86       - A Profile for Windows 2003 SP0 x86
Win2003SP1x64       - A Profile for Windows 2003 SP1 x64
Win2003SP1x86       - A Profile for Windows 2003 SP1 x86
Win2003SP2x64       - A Profile for Windows 2003 SP2 x64
Win2003SP2x86       - A Profile for Windows 2003 SP2 x86
Win2008R2SP0x64     - A Profile for Windows 2008 R2 SP0 x64
Win2008R2SP1x64     - A Profile for Windows 2008 R2 SP1 x64
Win2008SP1x64       - A Profile for Windows 2008 SP1 x64
Win2008SP1x86       - A Profile for Windows 2008 SP1 x86
Win2008SP2x64       - A Profile for Windows 2008 SP2 x64
Win2008SP2x86       - A Profile for Windows 2008 SP2 x86
Win7SP0x64          - A Profile for Windows 7 SP0 x64
Win7SP0x86          - A Profile for Windows 7 SP0 x86
Win7SP1x64          - A Profile for Windows 7 SP1 x64
Win7SP1x86          - A Profile for Windows 7 SP1 x86
WinXPSP1x64         - A Profile for Windows XP SP1 x64
WinXPSP2x64         - A Profile for Windows XP SP2 x64
WinXPSP2x86         - A Profile for Windows XP SP2 x86
WinXPSP3x86         - A Profile for Windows XP SP3 x86
```

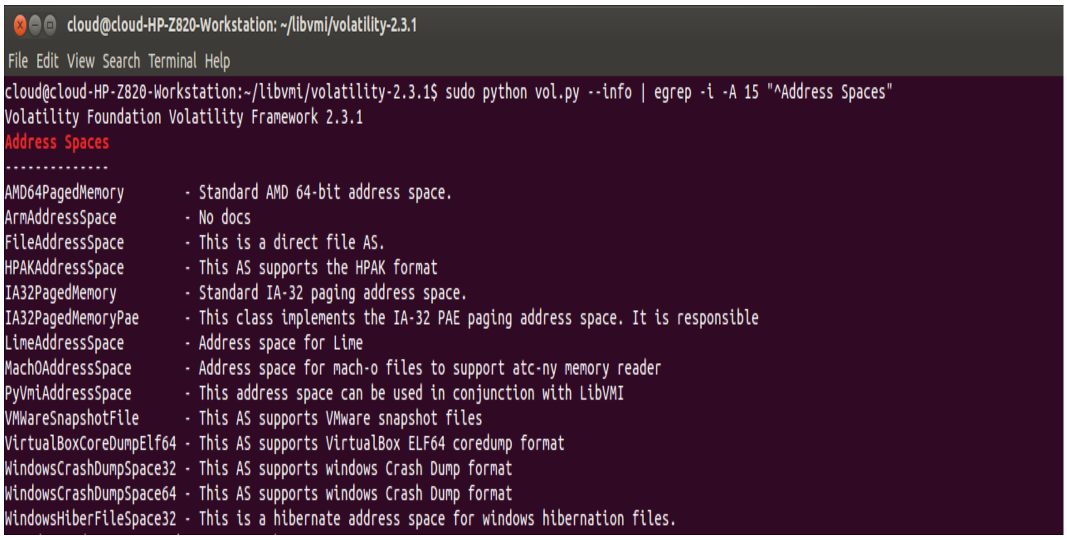
FIGURE 7.8: Supported Profile List in Our KVM Platform

For Windows guest memory analysis, no additional steps are necessary to generate the corresponding profile. This is not the case for Linux. Since there exists all kinds of Linux distribution and Linux's kernel is always in constant evolution, we need to create a unique profile for every kernel version (2.6.x, 3.x, etc.), every distribution (such as Ubuntu/Fedora/CentOS,etc.). Plenty of online tutorials are available for this subject, for example: `https://code.google.com/p/volatility/wiki/LinuxMemoryForensics`.

7.4.5 Address Space

The address space notion is used to describe different memory dump. For different memory dump format, corresponding address space should be used to assure the correct functioning of memory analysis. It's not necessary to indicate which address space is used when calling a certain Volatility command. Volatility applies a heuristic algorithm to automatically choose the appropriate address space for input memory dump. The following figure has shown all support address space in our KVM virtualization platform.

The address space could be also extended by developing special plugin for special memory dump. For example, to make Volatility to support LibVMI's API, the author of LibVMi



```
cloud@cloud-HP-Z820-Workstation: ~/libvmi/volatility-2.3.1
File Edit View Search Terminal Help
cloud@cloud-HP-Z820-Workstation:~/libvmi/volatility-2.3.1$ sudo python vol.py --info | egrep -i -A 15 "^Address Spaces"
Volatility Foundation Volatility Framework 2.3.1
Address Spaces
--------------
AMD64PagedMemory          - Standard AMD 64-bit address space.
ArmAddressSpace           - No docs
FileAddressSpace          - This is a direct file AS.
HPAKAddressSpace          - This AS supports the HPAK format
IA32PagedMemory           - Standard IA-32 paging address space.
IA32PagedMemoryPae        - This class implements the IA-32 PAE paging address space. It is responsible
LimeAddressSpace          - Address space for Lime
MachOAddressSpace         - Address space for mach-o files to support atc-ny memory reader
PyVmiAddressSpace         - This address space can be used in conjunction with LibVMI
VMWareSnapshotFile        - This AS supports VMware snapshot files
VirtualBoxCoreDumpElf64 - This AS supports VirtualBox ELF64 coredump format
WindowsCrashDumpSpace32 - This AS supports windows Crash Dump format
WindowsCrashDumpSpace64 - This AS supports windows Crash Dump format
WindowsHiberFileSpace32 - This is a hibernate address space for windows hibernation files.
```

FIGURE 7.9: Volatility Supported Address Space

has developed a special address space plugin called "PyVmiAddressSpace". It's only necessary to copy Python script "pyvmiaddressspace.py" into the following directory under Volatility 2.3: volatility/plugins/addrspaces/.

7.5 Cooperation with LibVMI

Volatility is designed to work on forensic memory snapshots. In this mode, a forensic analyst would take a physical memory image from a target machine, and then use Volatility to extract useful information from that image. However, since Volatility already contains significant information on the Windows/Linux memory layout and because Volatility greatly simplifies the development of memory analysis tools, LibVMI development team tried to integrate Volatility with LibVMI to facilitate analysis on a running virtual machine [17]. Thanks to their great job, now a "VMI application-LibVMI-Volatility" tool chain has been established to simplify the development of VMI application. In this section, we talk about how they make Volatility work directly on a live virtual machine.

In fact, LibVMI is implemented in C and we should develop some VMI applications (out-of-band) with its C library. With the popularity of Python in VMI research domain, LibVMI also provides a Python wrapper called PyVMI to allow LibVMI being used by Python script, such as Volatility framework. Their effort has formed the software stack shown in Figure 7.10.

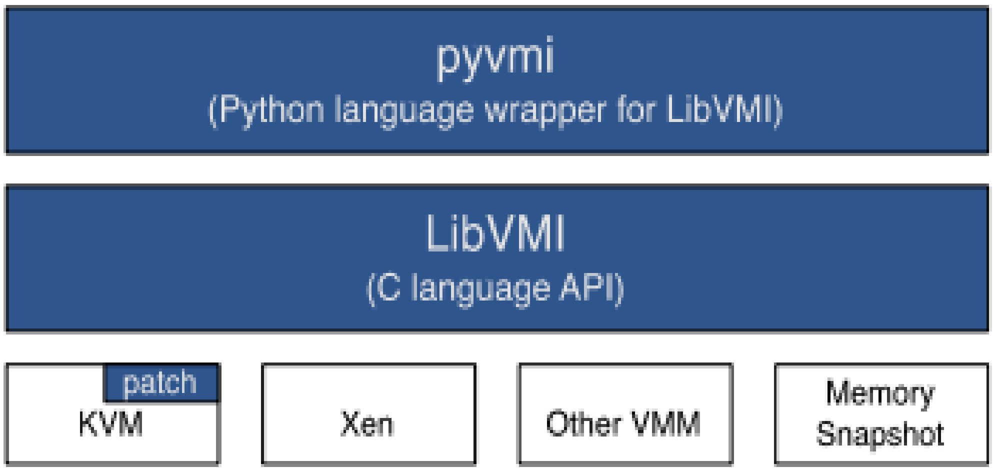


FIGURE 7.10: Software stack with PyVMI wrapper on top of the C language LibVMI library [1]

With Pyvmi, LibVMI could be used a module in Python script. Now, we should consider how to allow Volatility functioning on a live virtual machine. To do this, LibVMI provides two mechanisms.

7.5.1 PyVMI Address Space Plugin

This first mechanism is using a special address space plugin called "PyVMIAddressSpace". Figure 7.11 shows how Volatility plugins could leverage this address space plugin to cooperate with LibVMI. Supposing we want to investigate which processes are currently running in target guest named "win7_32bit", this command could help us:

```
# sudo python path/to/vol.py pslist -l vmi:///win7_32bit --profile=Win7SP1x86
```

Although LibVMI development team announced that with PyVMI address space plugin, all Volatility plugins will work on a running virtual machine. With my manipulation we encountered some unexpected problems. Firstly, this plugin works for almost all Windows plugin while plants for some Linux plugins such as linux_netstat. Secondly, sometimes (not always) after applying for example netscan plugin for a Windows, I found that the volume of log file (under path /var/log/libvirt/qemu) possibly augmented until all root file system's disk space was all consumed. To solve this problem, I have posed this problem in LibVMI google discussion group but not received any response. I don't know this problem is specific to my KVM platform. Therefore, at least in my work environment, PyVMI address space plugin is not recommended compared to its alternative.

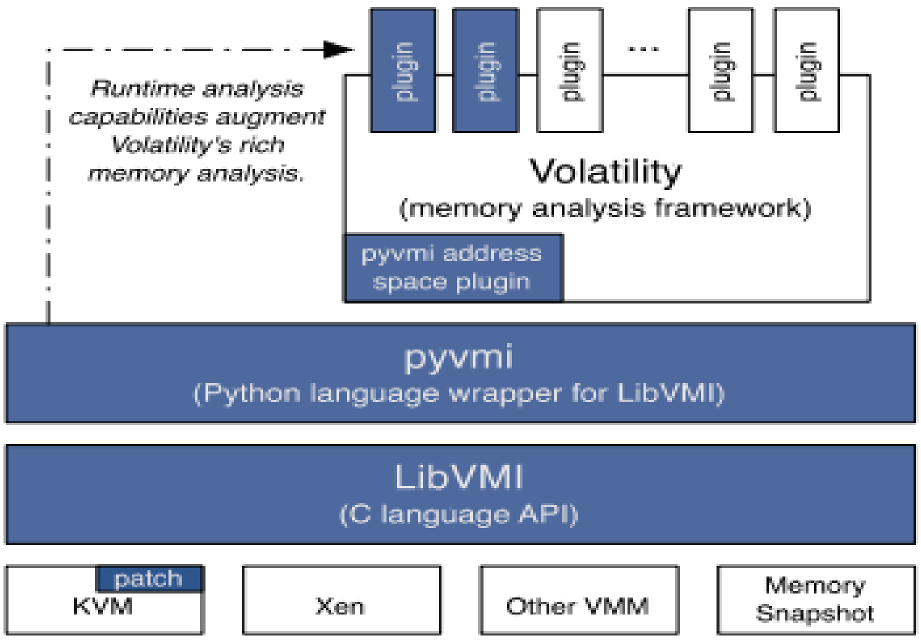

plugin
plugin
plugin
...
plugin
plugin
Runtime analysis capabilities augment Volatility's rich memory analysis.
Volatility
(memory analysis framework)
pyvmi address space plugin
pyvmi
(Python language wrapper for LibVMI)
LibVMI
(C language API)
patch
KVM
Xen
Other VMM
Memory Snapshot

FIGURE 7.11: Software stack with Volatility address space plugin [1]

7.5.2 Pyvmifs.py script

The second mechanism is firstly mounting virtual machine's physical memory as a regular file then using Volatility to analyze this memory file as if it were static. LibVMI has provided a special Python script to achieve this task. Its usage is:

```
# sudo python path/to/pyvmifs.py -o allow_other -o domain=GUEST_NAME path/to/mount/point
```

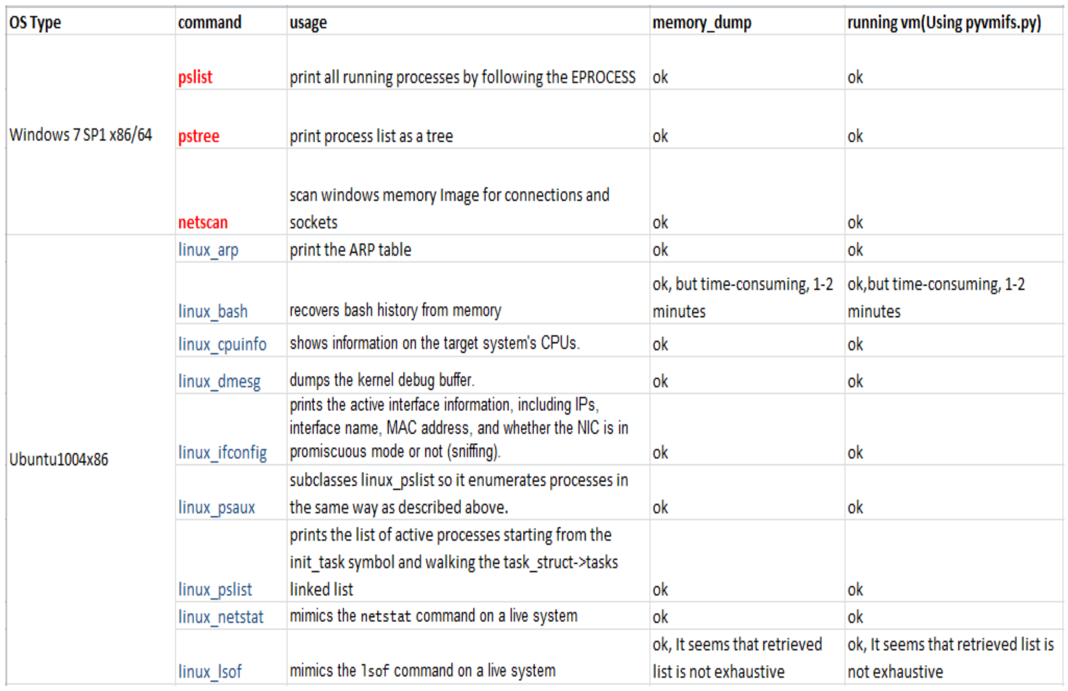
Still take our Windows guest "win7_32bit" as example. After above invocation, its physical memory is mounted under path /tmp/win7_32bit/mem. Then invoke for example pslist plugin to get the running process list:

```
# sudo python path/to/vol/py pslist -f /tmp/win7_32bit/mem --profile=Win7SP1x86
```

Different with pyvmi address space plugin, the simple Python script works well and poses none problem.

7.6 Experimentation Result

With LibVMI+pyvmifs.py+volatility, we could run all interesting plugins for our KVM Windows/Linux virtual machines. Figure shows the resume of execution of under Windows 7 x86/64 and Ubuntu 10.04 x86.



OS Type	command	usage	memory_dump	running vm(Using pyvmifs.py)
Windows 7 SP1 x86/64	pslist	print all running processes by following the EPROCESS	ok	ok
	pstree	print process list as a tree	ok	ok
	netscan	scan windows memory Image for connections and sockets	ok	ok
Ubuntu1004x86	linux_arp	print the ARP table	ok	ok
	linux_bash	recovers bash history from memory	ok, but time-consuming, 1-2 minutes	ok,but time-consuming, 1-2 minutes
	linux_cpuinfo	shows information on the target system's CPUs.	ok	ok
	linux_dmesg	dumps the kernel debug buffer.	ok	ok
	linux_ifconfig	prints the active interface information, including IPs, interface name, MAC address, and whether the NIC is in promiscuous mode or not (sniffing).	ok	ok
	linux_psaux	subclasses linux_pslist so it enumerates processes in the same way as described above.	ok	ok
	linux_pslist	prints the list of active processes starting from the init_task symbol and walking the task_struct->tasks linked list	ok	ok
	linux_netstat	mimics the netstat command on a live system	ok	ok
	linux_lsof	mimics the lsof command on a live system	ok, It seems that retrieved list is not exhaustive	ok, It seems that retrieved list is not exhaustive

Figure 7.12: Experimentation Result of Volatility Plugins in KVM Platform

7.7Limit of Volatility&LibVMI

Although Volatility presents various advantages in memory forensic analysis domain, its disadvantages are still obvious in the context of Virtual Machine Introspection: Volatility is applied uniquely to memory analysis, whereas Virtual Machine Introspection could also cover introspection on virtual CPU or virtual disk. Fortunately, LibVMI and libguestfs [24] could come to cover this shortage. LibVMI allows monitoring the running state of vCPU and libguestfs library is able to access and modify virtual machine's disk image.

Chapter 8

VIRTUOSO INSTALLATION AND MANIPULATION

Although forensic memory analysis community has developed plenty of utilities which could be leveraged for VM introspection, VMI applications based on these utilities such as volatility plugin, are still not able to be applied conveniently in for example operator provided cloud environment. The reason is obvious: the approach strongly depends on the stability of target VM kernel and need much more humain effort. Once update or patch is applied to monitored guest's kernel, these VMI applications may need to be adapted. To remedy this mentioned problem, we need to think out of box. An interesting and fundamental insight is: it is typically trivival to write programs inside the guest OS that compute the desired information by querying the built-in APSs. Logcially and naturally, people think to how to reutulize the existing API(binary code) in the guest to automate the generation of VMI tools. Not only the binary code, the execution contex under some circumstance also could be leveraged to do VMI, we call this philosophy as "Reutilization Pattern".

Nowdays there exist some famous and attractive research efforts based on reutilizatin pattern, such as VIRTUOSO[11], EXTERIOR[20], HyperShell[22],etc.. Among all these VMI techonologies, HyperShell is the most attractive: it provides a hypervisor layer guest OS shell that has all of the functionalities of a traditional shell, but offers better automation, uniformity and centralized management. However, its source code is not accessible, but I still recommand to give it some attention. Because VIRTUOSO is the unique open source tool in this domain, therefore, we still have to explore VIRTUOSO to hava a more depper knowledge about reutilization pattern.

8.1 Implementation of VIRTUOSO

To illustrate our disscusion about VIRTUOSO(installation, execution and assesse), it is better to have a general introduction about its design and implementation. As we talked before, VIRTUOSO's objective is how to translate a guest-OS program into a form that could run outside of its native environment. To achevie this, VIRTUOSO relies on dynamic analysis to capture all the codes executed while the target program is runing. Meanwhile, VIRTUOSO leverages dynamic slicing technique to identify the exact set of instructions required to compute the introspection. In addition, due to the dynamic analysis nature of VIRTUOSO, it is supposed to have a trace merging algorithm to merge the gathered traces from multiple time execution. In one word, VIRUOSO's innovation lies in three key techniques : dynamic analysis, dynamic slicing and trace emerging.

Generally speaking, VIRTUOSO creates introspection tools for an operating system by converting in-guest programs, which query guest OS's public APIs, into hypervisor-level programs that reproduce the almost same result with in-guest utilises. This process could be finished in three phases: training phase, analysis phase and runtime phase.

As shown in Figure 8.1, training phase is conducted by ”Trace Logger” component in a trusted VM , which has the same OS and kernel version with clients' OS. Trace logger is acutally an modified version of QEMU 0.9.1 in which logging all instructions functionality is added. the training programs running in trusted guest are in charge of to signal the Trace Logger and inform the latter of the beginning and end of the introspection operation. Thus, we could infer that writing a correct training program is not trivival.

The gathered traces in training phase contain the instructions of the whole system, namely, in addtion to computing the introspection quantity , the traces also include unrelated events such as interrupt handling. we nened to excise those extraneous parts of the traces. This is the objective of component ”Trace Analyser”. This anaylsis depends on dynamic data slice techinique applied on each trace. Finally, we merge the slice results accross basic blocks and traces, producing a unified program that could be translated into an out-of-guest intro routine. In pracice, this component is written in Python.

The translated code could not be executed directly, il must be called with an appropriate runtime environment in which to execute. Currently, the generated intropsection utilities are in form of Volatility plugin. Therefore, the runtime environment in current implementation of VIRTUOSO is Volatility.

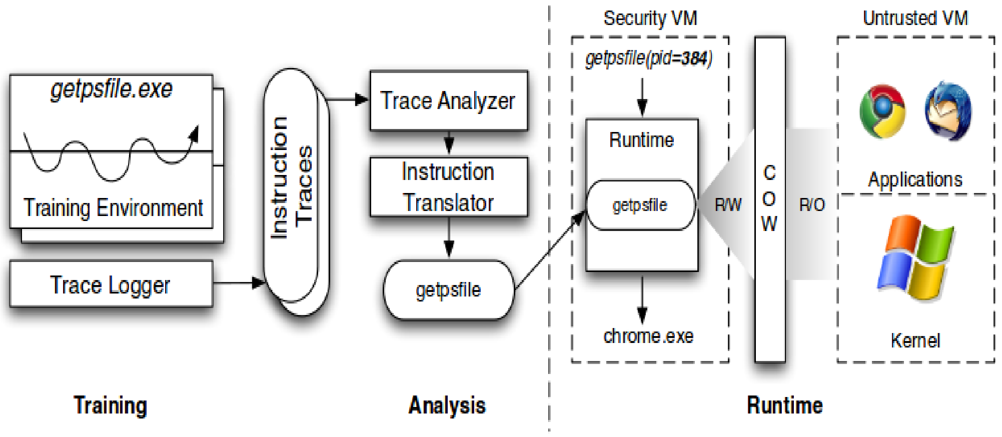


FIGURE 8.1: VIRTUOSO's Component and Architecture [11]

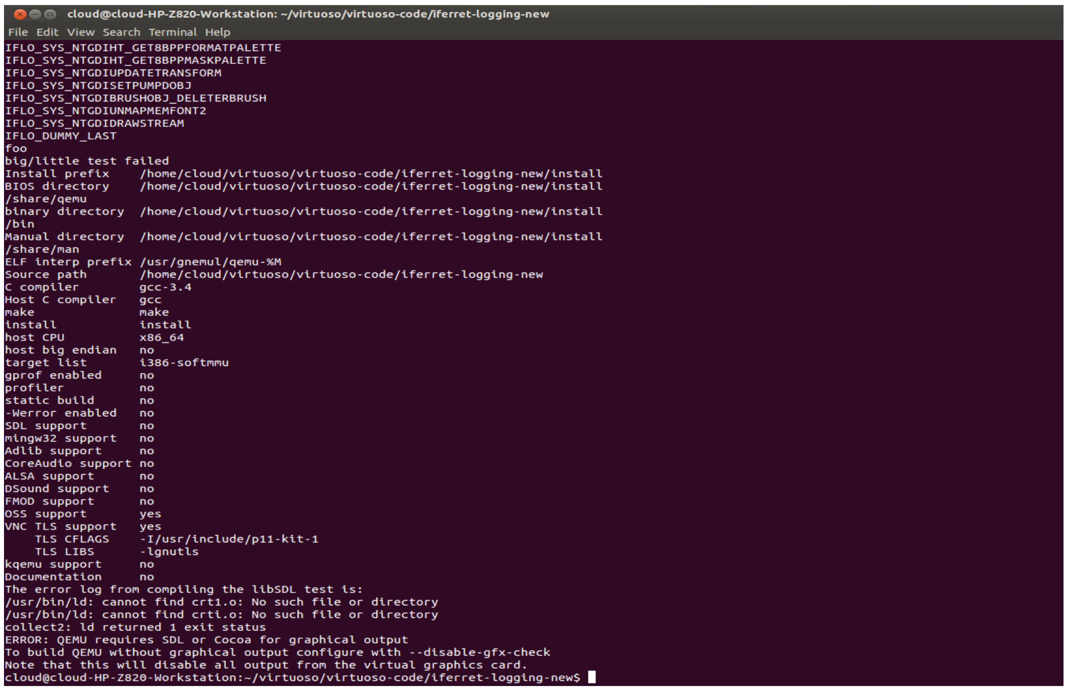
8.2 Installation of VIRTUOSO

A rather detailed and helpful how-to wiki about installation of VIRTUOSO is available in the link `https://code.google.com/p/virtuoso/wiki/Installation`. Since that this wiki has not seen any update since 2012, we were still stuck by some unexpected problems. As a meaningful complement to the initial how-to wiki, this tutorial aims to record all crossed problems and its solution in the road of VIRTUOSO exploration.

To install VIRTUOSO, 64-bit Linux system (Debian/Ubuntu preferred) and Python 2.6 or later are required. Its source code is available at `http://code.google.com/p/virtuoso/downloads/list`. After obtaining the VIRTUOSO source code, it consists of two steps to install VIRTUOSO: install all dependencies for VIRTUOSO (gcc 3.4, QEMU 0.9.1, libdasm), compile and install iFerret.

8.2.1 Problem 1: No SDL support for installation of QEMU 0.9.1

The first obstacle encountered appears during the installation of QEMU 0.9.1, which is shown in Figure 8.2.


cloud@cloud-HP-Z820-Workstation: ~/virtuoso/virtuoso-code/iferret-logging-new
File Edit View Search Terminal Help

```
IFLO_SYS_NTGDIHT_GET8BPPFORMATPALETTE
IFLO_SYS_NTGDIHT_GET8BPPMASKPALETTE
IFLO_SYS_NTGDIUPDATETRANSFORM
IFLO_SYS_NTGDISETPUMPDOBJ
IFLO_SYS_NTGDIBRUSHOBJ_DELETERBRUSH
IFLO_SYS_NTGDIUNMAPMEMFONT2
IFLO_SYS_NTGDIDRAWSTREAM
IFLO_DUMMY_LAST
foo
big/little test failed
Install prefix    /home/cloud/virtuoso/virtuoso-code/iferret-logging-new/install
BIOS directory    /home/cloud/virtuoso/virtuoso-code/iferret-logging-new/install
/share/qemu
binary directory  /home/cloud/virtuoso/virtuoso-code/iferret-logging-new/install
/bin
Manual directory  /home/cloud/virtuoso/virtuoso-code/iferret-logging-new/install
/share/man
ELF interp prefix /usr/gnemul/qemu-%M
Source path       /home/cloud/virtuoso/virtuoso-code/iferret-logging-new
C compiler        gcc-3.4
Host C compiler   gcc
make              make
install           install
host CPU          x86_64
host big endian   no
target list       i386-softmmu
gprof enabled     no
profiler          no
static build      no
-Werror enabled   no
SDL support       no
mingw32 support   no
Adlib support     no
CoreAudio support no
ALSA support      no
DSound support    no
FMOD support      no
OSS support       yes
VNC TLS support   yes
    TLS CFLAGS    -I/usr/include/p11-kit-1
    TLS LIBS      -lgnutls
kqemu support     no
Documentation     no
The error log from compiling the libSDL test is:
/usr/bin/ld: cannot find crt1.o: No such file or directory
/usr/bin/ld: cannot find crti.o: No such file or directory
collect2: ld returned 1 exit status
ERROR: QEMU requires SDL or Cocoa for graphical output
To build QEMU without graphical output configure with --disable-gfx-check
Note that this will disable all output from the virtual graphics card.
cloud@cloud-HP-Z820-Workstation:~/virtuoso/virtuoso-code/iferret-logging-new$ 
```

FIGURE 8.2: No SDL support for installation of QEMU 0.9.1

The cause of this error lies in that system could not find graphical output support for QEMU 0.9.1(notice that SDL support check's result is no.). After some searching online, I found a solution to this problem: It should issue the following command in the terminal a prior to installing iFerret component of VIRTUSO:

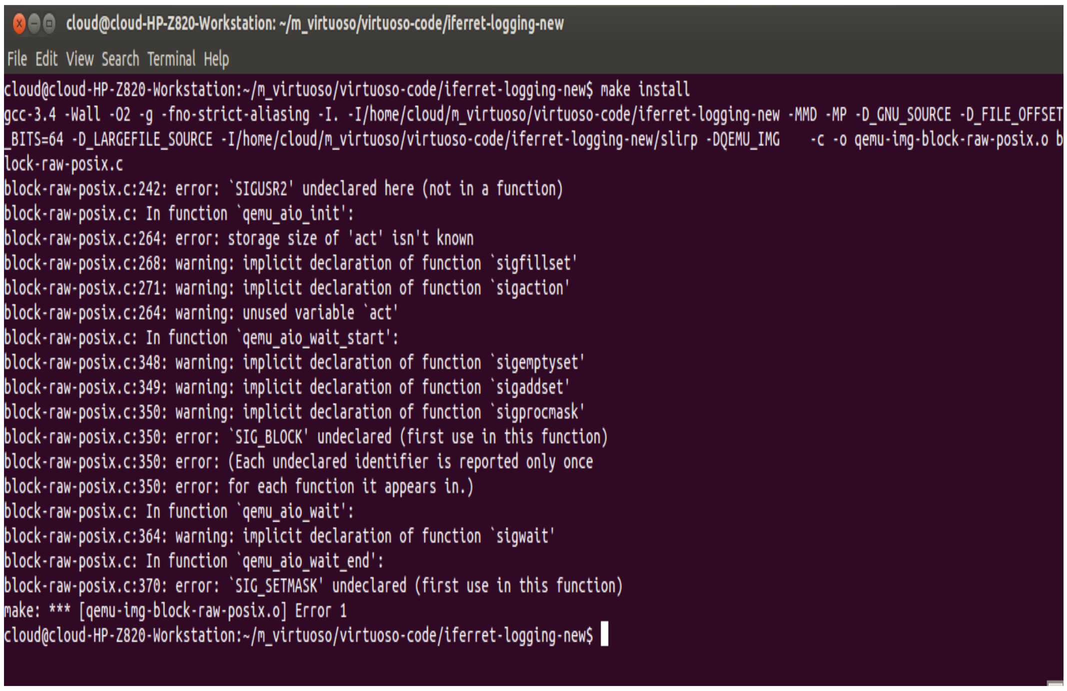
```
# export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu
```

With this export command, system is able to find the path to libSDL library required by QEMU 0.9.1.

8.2.2 Problem 2: iFerret compile error

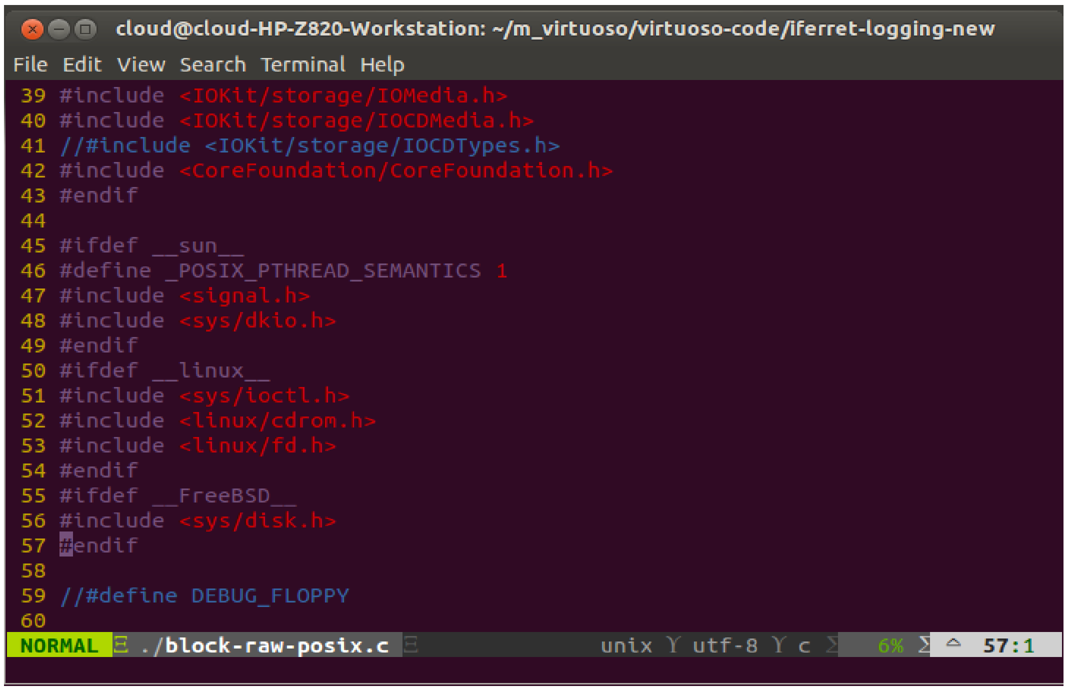
The error is occurred when we change into the 'iferret-logging-new' directory and issue 'make install' command, as shown in Figure 8.3.

Obviously, from above figure, we could infer that there exist some bugs in source file 'block-raw-posix.c'. After some investigation, this is caused by a macro definition syntax. Look at the block of code shown in Figure 8.4 which causes the above error:


cloud@cloud-HP-Z820-Workstation: ~/m_virtuoso/virtuoso-code/iferret-logging-new
File Edit View Search Terminal Help

```
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/iferret-logging-new$ make install
gcc-3.4 -Wall -O2 -g -fno-strict-aliasing -I. -I/home/cloud/m_virtuoso/virtuoso-code/iferret-logging-new -MMD -MP -D_GNU_SOURCE -D_FILE_OFFSET
_BITS=64 -D_LARGEFILE_SOURCE -I/home/cloud/m_virtuoso/virtuoso-code/iferret-logging-new/slirp -DQEMU_IMG    -c -o qemu-img-block-raw-posix.o b
lock-raw-posix.c
block-raw-posix.c:242: error: `SIGUSR2' undeclared here (not in a function)
block-raw-posix.c: In function `qemu_aio_init':
block-raw-posix.c:264: error: storage size of 'act' isn't known
block-raw-posix.c:268: warning: implicit declaration of function `sigfillset'
block-raw-posix.c:271: warning: implicit declaration of function `sigaction'
block-raw-posix.c:264: warning: unused variable `act'
block-raw-posix.c: In function `qemu_aio_wait_start':
block-raw-posix.c:348: warning: implicit declaration of function `sigemptyset'
block-raw-posix.c:349: warning: implicit declaration of function `sigaddset'
block-raw-posix.c:350: warning: implicit declaration of function `sigprocmask'
block-raw-posix.c:350: error: `SIG_BLOCK' undeclared (first use in this function)
block-raw-posix.c:350: error: (Each undeclared identifier is reported only once
block-raw-posix.c:350: error: for each function it appears in.)
block-raw-posix.c: In function `qemu_aio_wait':
block-raw-posix.c:364: warning: implicit declaration of function `sigwait'
block-raw-posix.c: In function `qemu_aio_wait_end':
block-raw-posix.c:370: error: `SIG_SETMASK' undeclared (first use in this function)
make: *** [qemu-img-block-raw-posix.o] Error 1
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/iferret-logging-new$
```

Figure 8.3: iFerret compile error


cloud@cloud-HP-Z820-Workstation: ~/m_virtuoso/virtuoso-code/iferret-logging-new
File Edit View Search Terminal Help

```
#include <IOKit/storage/IOMedia.h>
#include <IOKit/storage/IOCDMedia.h>
//#include <IOKit/storage/IOCDTypes.h>
#include <CoreFoundation/CoreFoundation.h>
#endif

#ifdef __sun__
#define _POSIX_PTHREAD_SEMANTICS 1
#include <signal.h>
#include <sys/dkio.h>
#endif
#ifdef __linux__
#include <sys/ioctl.h>
#include <linux/cdrom.h>
#include <linux/fd.h>
#endif
#ifdef __FreeBSD__
#include <sys/disk.h>
#endif

//#define DEBUG_FLOPPY

NORMAL  ./block-raw-posix.c          unix  utf-8  c   6%   57:1
```

Figure 8.4: Bloc of code causing compile error

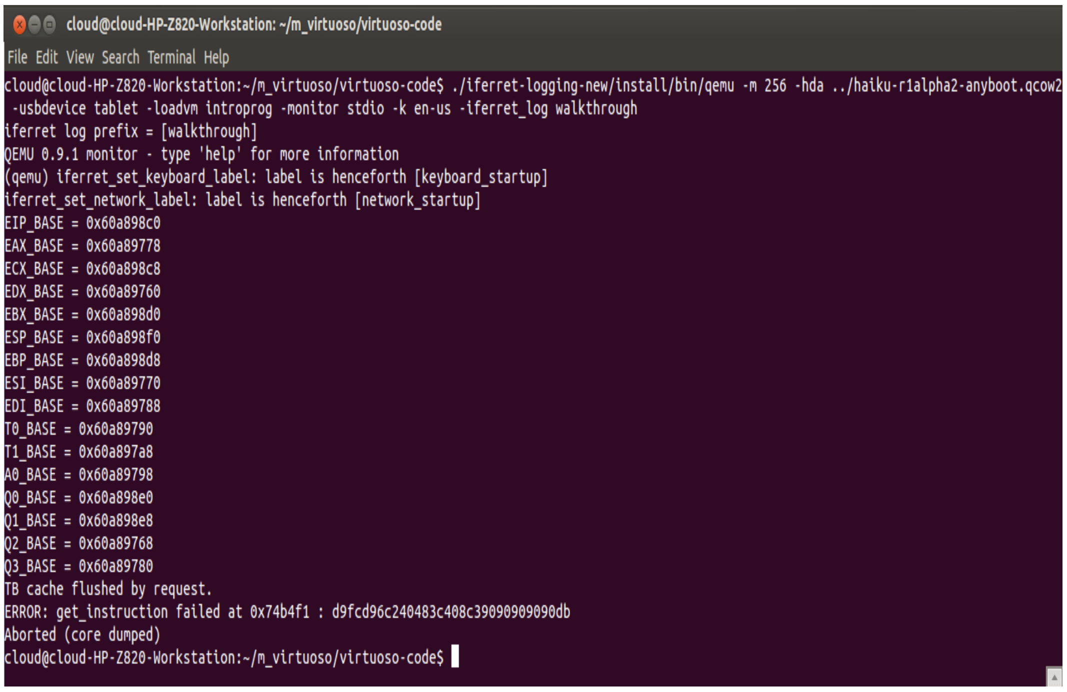
C header file 'signal.h' is included under some conditions. To solve this problem, it simply needs to add one line instruction at the beginning of this file: #include ¡signal.h¿. Repeat 'make install' command, normally, no errors will appear and the installation process should be finished with success. Now you could give a try to VIRTUOSO following another wiki in its official site.

8.2.3 Problem 3: The abort of execution of example VM image

We follow this link `https://code.google.com/p/virtuoso/wiki/Walkthrough` to have a try with VIRTUOSO. Issuing the command line:

```
# install/bin/qemu -m 256 -hda haiku-r1alpha2-anyboot.qcow2 -usbdevice tablet

-loadvm introprog -monitor stdio -k en-us -iferret_log walkthrough
```

However, new error has occurred:


cloud@cloud-HP-Z820-Workstation: ~/m_virtuoso/virtuoso-code
File Edit View Search Terminal Help

```
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code$ ./iferret-logging-new/install/bin/qemu -m 256 -hda ../haiku-r1alpha2-anyboot.qcow2
 -usbdevice tablet -loadvm introprog -monitor stdio -k en-us -iferret_log walkthrough
iferret log prefix = [walkthrough]
QEMU 0.9.1 monitor - type 'help' for more information
(qemu) iferret_set_keyboard_label: label is henceforth [keyboard_startup]
iferret_set_network_label: label is henceforth [network_startup]
EIP_BASE = 0x60a898c0
EAX_BASE = 0x60a89778
ECX_BASE = 0x60a898c8
EDX_BASE = 0x60a89760
EBX_BASE = 0x60a898d0
ESP_BASE = 0x60a898f0
EBP_BASE = 0x60a898d8
ESI_BASE = 0x60a89770
EDI_BASE = 0x60a89788
T0_BASE = 0x60a89790
T1_BASE = 0x60a897a8
A0_BASE = 0x60a89798
Q0_BASE = 0x60a898e0
Q1_BASE = 0x60a898e8
Q2_BASE = 0x60a89768
Q3_BASE = 0x60a89780
TB cache flushed by request.
ERROR: get_instruction failed at 0x74b4f1 : d9fcd96c240483c408c39090909090db
Aborted (core dumped)
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code$
```

FIGURE 8.5: Abort of execution of example VM image

This error is not evident as others obstacles we encountered before and nothing could be found online. After one week's effort, I finally got some clues about this problem to read another wiki 'limitation' at `https://code.google.com/p/virtuoso/wiki/Limitations`, This is caused by the fact that:

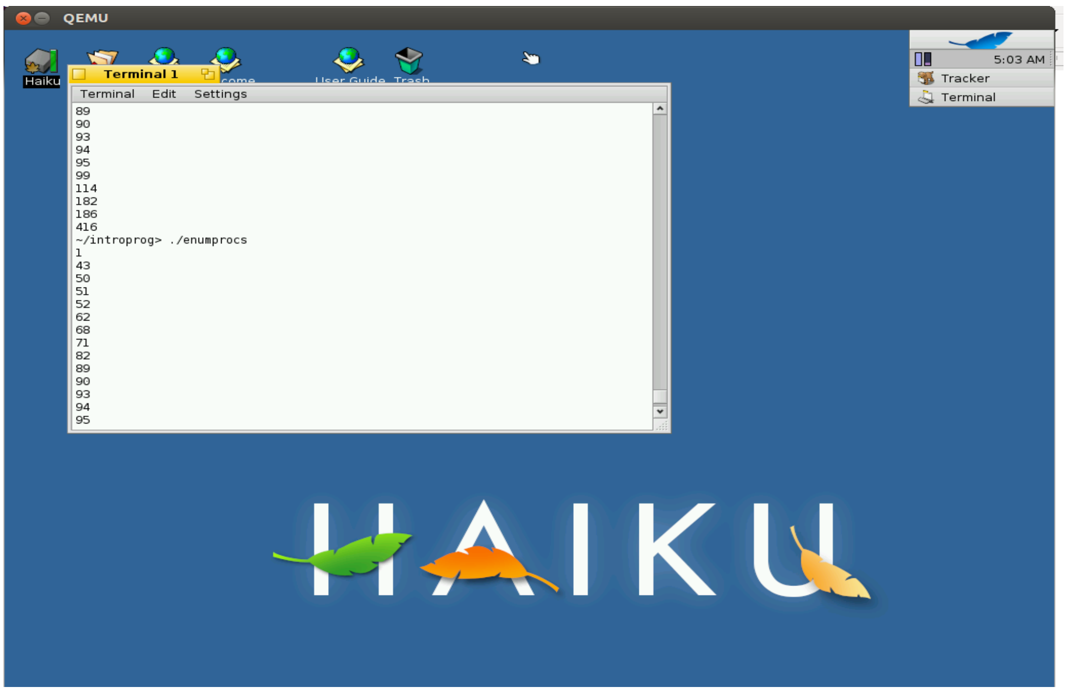
"Virtuoso makes use of libdasm to disassemble instructions (see iferret-logging-new/target-i386/translate.c for details). Libdasm is missing support for some instructions, and this will cause tracing to stop and QEMU to shut down. The disassembly Virtuoso does is mainly for debugging, and so these lines can be commented out if they cause trouble. I would like to switch to a more reliable disassembler in the future, however."

Therefore, the answer to the problem is: libdasm encountered some unknown instructions in Ubuntu14.04 and forces VIRTUOSO to abort. To solve this, we need to identify and comment related block of code in file 'translate.c' and recompile all. Concretely, comment line from 3281 to 3335 in file 'iferret-logging-new/target-i386/translator.c' and recompile install all.

Until now, we could take a ride with VIRTUOSO. Change into directory 'iferret-logging-new' and issue the following command line into a terminal:

```
# install/bin/qemu -m 256 -hda haiku-r1alpha2-anyboot.qcow2 -usbdevice tablet
-loadvm introprog -monitor stdio -k en-us -iferret_log walkthrough
```

Some explanation about above command: To prove that VIRTUOSO is OS-angostic VMI technology, we use a Haiku OS-based virtual machine image provided by author. This virtual machine comes with a snapshot named "introprog" that has a few training programs already loaded and compiled. For example, we could run a training program named "enumprocs" to get the PID list of currently running process in guest virtual machine. The execution result is shown in Figure 8.6.



```
89
90
93
94
95
99
114
182
186
416
~/introprog> ./enumprocs
1
43
50
51
52
62
68
71
82
89
90
93
94
95
```

FIGURE 8.6: Run Training program in trusted VM

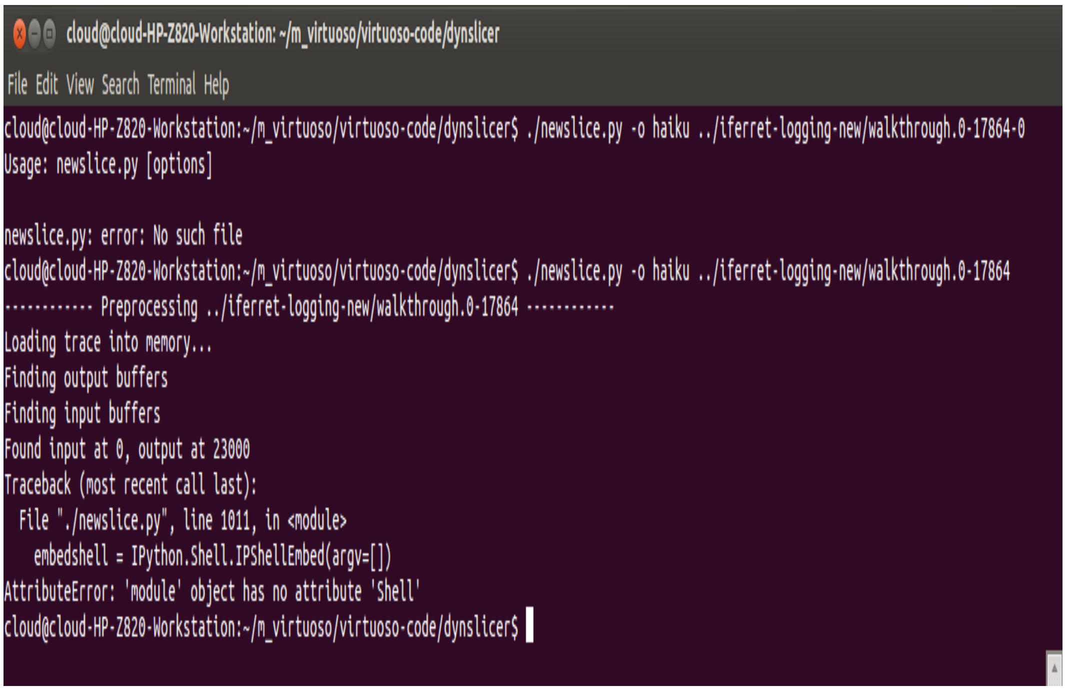
Notice that, after several times of invocation of small programs such as 'enumprocs', we observe that some execution traces files are generated. Generated output file will be placed in directory where VIRTUOSO is called. (In this case, it is directory 'iferret-logging-new')

8.2.4 Problem 4: IPython runtime exception when generating inspection code with VIRTUOSO

Given that, component 'iFerret' of VIRTUOSO has obtained execution traces of training program, it is time to analysis trace file and produces the Volatility plugin. To achieve this, go to the dynslicer directory and run:

```
# ./newslicer.py -o haiku ../iferrret-logging-new/walkthrough.0-17864
```

Unfortunately, there comes a new error:



cloud@cloud-HP-Z820-Workstation: ~/m_virtuoso/virtuoso-code/dynslicer
File Edit View Search Terminal Help

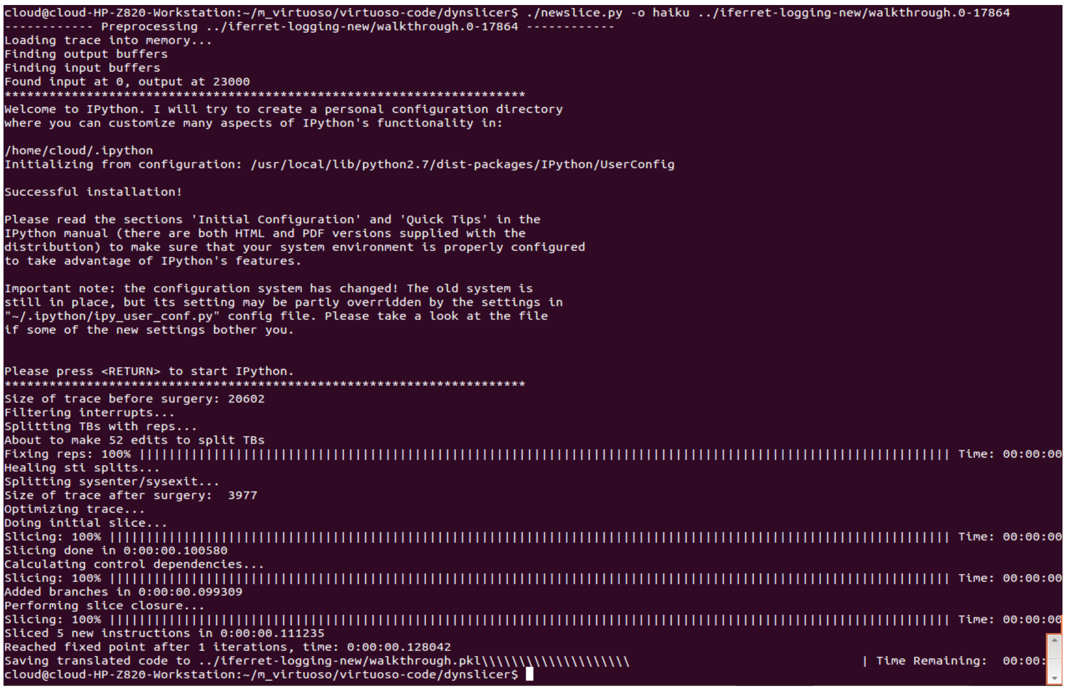
```
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/dynslicer$ ./newslice.py -o haiku ../iferret-logging-new/walkthrough.0-17864-0
Usage: newslice.py [options]

newslice.py: error: No such file
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/dynslicer$ ./newslice.py -o haiku ../iferret-logging-new/walkthrough.0-17864
----------- Preprocessing ../iferret-logging-new/walkthrough.0-17864 -----------
Loading trace into memory...
Finding output buffers
Finding input buffers
Found input at 0, output at 23000
Traceback (most recent call last):
  File "./newslice.py", line 1011, in <module>
    embedshell = IPython.Shell.IPShellEmbed(argv=[])
AttributeError: 'module' object has no attribute 'Shell'
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/dynslicer$
```

FIGURE 8.7: IPython run exception

Definitely, this is related to the IPython version installed in our workstation. A possible solution to this problem is:

```
# sudo pip uninstall ipython

# sudo pip --proxy=http://proxy.rd.francetelecom.fr:8080 install ipython==0.10
```

Then re-execute above invocation, and output is shown in Figure 8.8. When Volatility



```
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/dynslicer$ ./newslice.py -o haiku ../iferret-logging-new/walkthrough.0-17864
------------ Preprocessing ../iferret-logging-new/walkthrough.0-17864 ------------
Loading trace into memory...
Finding output buffers
Finding input buffers
Found input at 0, output at 23000
**********************************************************************
Welcome to IPython. I will try to create a personal configuration directory
where you can customize many aspects of IPython's functionality in:

/home/cloud/.ipython
Initializing from configuration: /usr/local/lib/python2.7/dist-packages/IPython/UserConfig

Successful installation!

Please read the sections 'Initial Configuration' and 'Quick Tips' in the
IPython manual (there are both HTML and PDF versions supplied with the
distribution) to make sure that your system environment is properly configured
to take advantage of IPython's features.

Important note: the configuration system has changed! The old system is
still in place, but its setting may be partly overridden by the settings in
"~/.ipython/ipy_user_conf.py" config file. Please take a look at the file
if some of the new settings bother you.


Please press <RETURN> to start IPython.
**********************************************************************
Size of trace before surgery: 20602
Filtering interrupts...
Splitting TBs with reps...
About to make 52 edits to split TBs
Fixing reps: 100% |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| Time: 00:00:00
Healing sti splits...
Splitting sysenter/sysexit...
Size of trace after surgery:  3977
Optimizing trace...
Doing initial slice...
Slicing: 100% |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| Time: 00:00:00
Slicing done in 0:00:00.100580
Calculating control dependencies...
Slicing: 100% |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| Time: 00:00:00
Added branches in 0:00:00.099309
Performing slice closure...
Slicing: 100% |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| Time: 00:00:00
Sliced 5 new instructions in 0:00:00.111235
Reached fixed point after 1 iterations, time: 0:00:00.128042
Saving translated code to ../iferret-logging-new/walkthrough.pkl\\\\\\\\\\\\\\\\\\\\                             | Time Remaining:  00:00:
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/dynslicer$
```

Figure 8.8: VIRTUOSO introspection tool generation

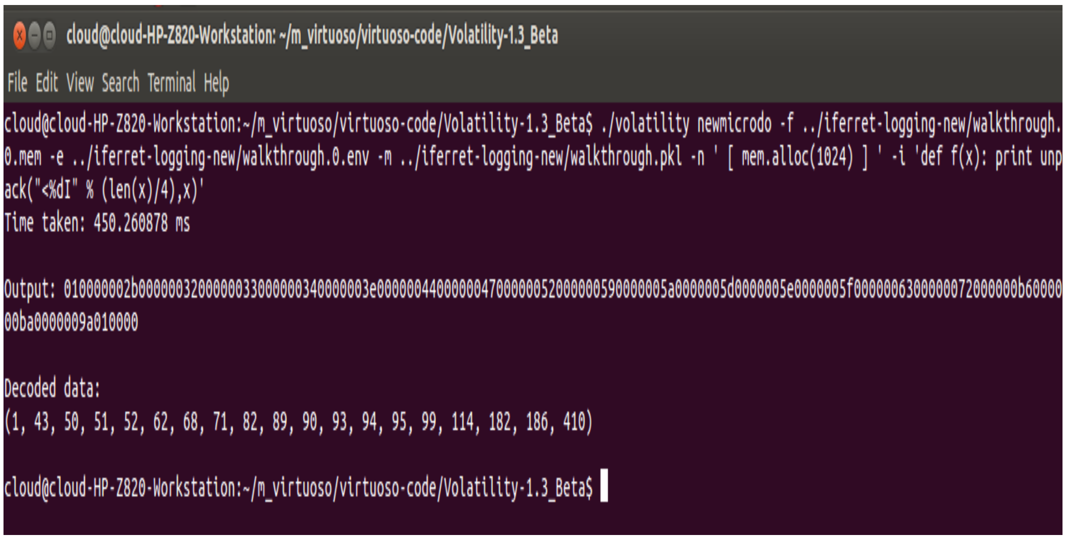
plugin is well generated, we could profit to realize introspection job at the level of hypervisor and get mostly the same view as inside the monitored VM. To do this, change into volatility-1.3-Beta directory and tape the following command:

```
# ./volatility newmicrodo -f ../iferret-logging-new/walkthrough.0.mem \

-e ../iferret-logging-new/walkthrough.0.env\

-m ../iferret-logging-new/walkthrough.pkl\

-n ' [ mem.alloc(1024) ] ' -i 'def f(x):  print unpack("<%dI" % (len(x)/4),x)'
```

8.3 Assessment of VIRTUOSO

Traditionally, the principal philosophy was to imitate the existing inspecting utility (ps, netstat command in Linux for example.) and rewrite the code from scratch with an intimate knowledge of OS kernel. This approach is rather intuitive but at the same time presents some obvious disadvantages:

- Prone to import new security when writing new introspection utilities
- Has to adapt each time target OS kernel has some updates


cloud@cloud-HP-Z820-Workstation: ~/m_virtuoso/virtuoso-code/Volatility-1.3_Beta
File Edit View Search Terminal Help

```
cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/Volatility-1.3_Beta$ ./volatility newmicrodo -f ../iferret-logging-new/walkthrough.
0.mem -e ../iferret-logging-new/walkthrough.0.env -m ../iferret-logging-new/walkthrough.pkl -n ' [ mem.alloc(1024) ] ' -i 'def f(x): print unp
ack("<%dI" % (len(x)/4),x)'
Time taken: 450.260878 ms

Output: 010000002b000000320000003300000034000000 3e000000440000004700000052000000590000005a0000005d0000005e0000005f000000630000007200000b60000
00ba0000009a010000

Decoded data:
(1, 43, 50, 51, 52, 62, 68, 71, 82, 89, 90, 93, 94, 95, 99, 114, 182, 186, 410)

cloud@cloud-HP-Z820-Workstation:~/m_virtuoso/virtuoso-code/Volatility-1.3_Beta$ 
```

Figure 8.9: VIRTUOSO introspection tool execution result

- Need much human effort

The greatest significance of VIRTUOSO lies in that it breaks away from conventions and leverages the reutilization of binary code. Exception the training stage, VIRTUOSO is capable of automatically generating introspection tool with obtained execution traces without target OS kernel knowledge.

However, the limitation of VIRTUOSO is also evident: first, it is not still completely automatic in generating introspection tool. VIRTUOSO still needs an expert to write the training programs to get system API execution traces in which we are interested. This expert is supposed to have an intimate knowledge about target system API, for example, which API is used to get the list of running processes in Haiku operating system. When writing training program, he should also assure that there is no process switch (context switch) during training program execution. In one word, it is a little difficult to write training program. Second, the range of introspection tools generated by VIRTUOSO depends on the target OS API, due to the fact that VIRTUOSO just simply reutilizes binary code s associated with system API. Third, VIRTUOSO is not able to extract device I/O code or instruction, VIRTUOSO is therefore not capable of generating network-side introspection tools. All above constraints hinder the application of VIRTUOSO in industry, but its philosophy inspired other interesting introspection tool, such as VMST.

Appendix A

CR3 Register Introduction

The CR3 register is a typical hardware anchor [3] where we could derive some OS-level information in the context of derivative pattern VMI. Since our study is about how to leverage derivative pattern in monitoring network traffic on per-process basis, it's better to resume the CR3's functionality and explore its potentiality for future usage. This is the motivation of this work and the remainder of this article is organized as follow. In the first part, some background conceptions about memory addressing, involved in the functioning of CR3 register, will be presented. Subsequently, we talk about how CR3's functionality specified in Intel developer manual [2] and helps in translating a linear address into a physical address. In the end, we address how CR3 register is leveraged to derive or infer system process information and its potentiality in network traffic monitoring.

A.1 Background [9]

To well understand the functionality of CR3 register, some basic and related conceptions about memory addressing should be kept in mind.

- **Logical Address**: consists of a segment and an offset, included in the machine language instructions to specify the address of an operand or an instruction.

- **Linear Address**: also known as virtual address, a single 32 bits unsigned integer that can be used to address up to 4GB. Their values range from 0x00000000 to 0xffffffff.
- **Physical Address**: used to address memory cell in memory chips.
- **MMU**: Memory Management Unit, a hardware circuit integrated in x86 architecture and it consist of two parts: Segmentation Unit helping CPU translate a logical address into a linear address and Paging Unit capable of transforming a linear address into a physical address.

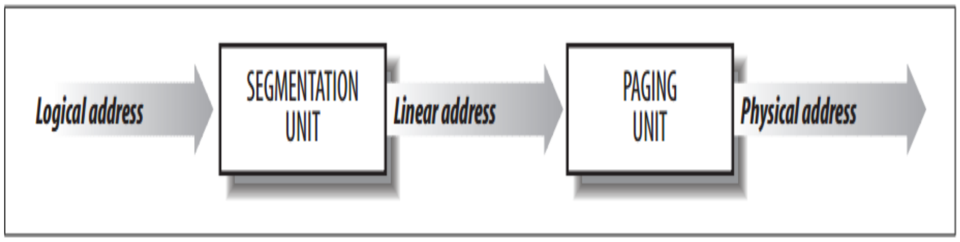


FIGURE A.1: Logical Address Translation [9]

Segmentation has been included in 80x86 microprocessors to encourage programmers to split their applications into logically related entities, such as subroutines or global and local data areas. However, Linux uses segmentation in a very limited way. In fact, segmentation and paging are somewhat redundant, because both can be used to separate the physical address spaces of processes: segmentation can assign a different linear address space to each process, while paging can map the same linear address space into different physical address spaces. Linux prefers paging to segmentation.

A.2 Functionality of CR3 Register [10]

CR3 register is a control register integrated in CPU. Along with other control registers such as CR0, CR1, CR2, CR3 and CR4, they are used to determine operating mode of the processor and the characteristics of the currently executing task. This section presents control register in the perspective of processors.

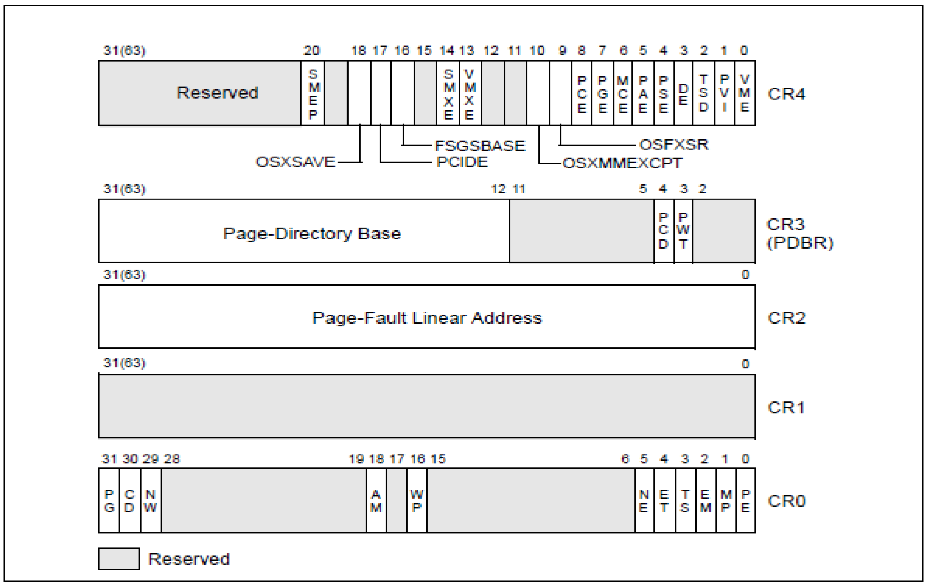


Figure A.2: Control Register Architecture [10]

Figure A.2 presents the four control registers' architecture. Their respective functionalities are summarized below, and each architecturally defined control field in these control registers are described individually.

- **CR0**: contains system control flags that control operating mode and states of the processor.
- **CR1**: reserved.
- **CR2**: contains the page-fault linear address (the linear address that caused a page fault).
- **CR3**: contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor's internal data caches (they do not control TLB caching of page-directory information). CR3 register works on condition that VIRTUAL ADDRESSING feature is enabled, which means that PG bit is set in CR0.
- **PCD**: Page-level Cache Disable (bit 4 of CR3) — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. This

bit is not used if paging is disabled, with PAE paging, or with IA-32e paging if CR4. PCIDE=1. Please refer to Section 4.9: "Paging and Memory Typing" in reference [] to get more information.

- **PWT**: Page-level Write-Through (bit 3 of CR3) — Controls the memory type used to access

A.3 Functioning of CR3 Register in Memory Addressing

Figure A.3 well explains how MMU uses CR3 register to translate a linear address into a physical one. The 32 bits linear address can be divided into 3 parts: Directory (the most significant 10 bits), Table (the intermediate 10 bits) and Offset (the least significant 12 bits). The physical address of the Page Directory in use is stored in a control register named CR3. The Directory field within the linear address determines the entry in the Page Directory that points to the proper Page Table. The address's Table field, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The offset field determines the relative position within the page frame.

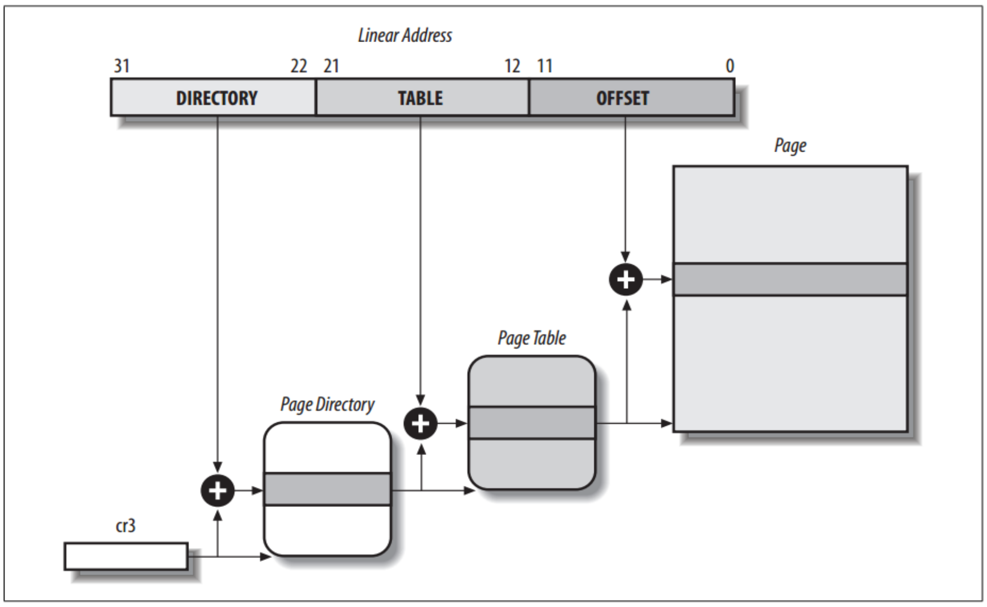


FIGURE A.3: Paging by 80x86 Processors

Since the CR3 register holds the address of the top-level Page Directory for the current context and a single address space is active per processor at any given time, the physical address stored in CR3 register could be used to represent its unique corresponding process. This is the theoretical base for that CR3 could be profited in the derivative pattern. For example, a new CR3 value means a new process has been created, a transition from one to another value in CR3 means that a context switch (process switch) has occurred. Antfarm [15] has leveraged this characteristic to enumerate all executing processes, monitor the creation, switch and exit of process. However, in Antfarm, cause of the limitation of derivative pattern, we could just get its page directory base address and executing time for each identified process. Due to the lack of semantic knowledge for kernel memory, it is impossible to get more detailed information (such as PID, process name, etc.) about identified processes. Based on Antfarm, Lycosid [18] whose objective is to detect the hidden malicious processes has been developed. Its idea is to compare a process list deduced by low-level hardware with that obtained by some system utility such as UNIX command "ps". In case of inconsistency about process number, Lycosid infers the presence of hidden processes in monitored VM and identify those processes in a statistical manner.

Bibliography

[1] B.D.Payne. Simplifying virtual machine introspection using libvmi. 2012. URL http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf.

[2] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[3] Bryan D. Payne and Wenke Lee. Secure and flexible monitoring of virtual machines. In *ACSAC*, pages 385–397. IEEE Computer Society.

[4] Volatility. . URL https://code.google.com/p/volatility/.

[5] Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093, pages 370–373. Springer, 2011.

[6] Christian Schneider, Jonas Pfoh, and Claudia Eckert. Bridging the semantic gap through static code analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM Press, 2012.

[7] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pages 39–58, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] What is kvm virtualization in linux?, . URL http://www.linuxnix.com/2013/02/what-is-kvm-virtualization-in-linux.html.

[9] Daniel Bovet and Marco Cesati. *Chapter 2: Memory Addressing, Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual, 2014.* Intel Corporation, 2014.

[11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. *2013 IEEE Symposium on Security and Privacy*, 0:297–312, 2011.

[12] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[13] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec '09)*, pages 1–10, Chicago, Illinois, USA, nov 2009. ACM Press.

[14] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 3–14, New York, NY, USA, 2014. ACM.

[15] Stephen T. Jones, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau. Antfarm: Tracking processes in a virtual machine environment. In *in Proc. of the USENIX Annual Technical Conf*, 2006.

[16] Jonas Pfoh. Leveraging derivative virtual machine introspection methods for security applications. 2013.

[17] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038, pages 96–112. Springer, nov 2011.

[18] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 91–100, New York, NY, USA, 2008. ACM.

[19] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection.

In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 586–600, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Yangchun Fu and Zhiqiang Lin. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. *SIGPLAN Not.*, 48(7):97–110, mar 2013.

[21] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 5:1–5:13, New York, NY, USA, 2012. ACM.

[22] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. Hypershell: A practical hypervisor layer guest os shell for automated in-vm management. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 85–96, Philadelphia, PA, Jun 2014. USENIX Association.

[23] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. pages 147–156. IEEE, 2011.

[24] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 545–554, New York, NY, USA, 2009.

[25] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2009)*, Fukuoka, Japan, March 2009.

[26] Lionel Litty and David Lie. Manitou: A layer-below approach to fighting malware. In *In Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID*, pages 6–11, 2006.

[27] Volatility, . URL `https://code.google.com/p/volatility/wiki/LinuxCommandReference23#linux_vma_cache`.

[28] Insight-vmi. . URL `https://code.google.com/p/insight-vmi/`.

[29] Ebtables community developers, . URL `http://ebtables.sourceforge.net/`.

[30] Libvirt community developers. . URL http://libvirt.org/goals.html.

[31] Libvirt wikipedia. . URL http://en.wikipedia.org/wiki/Libvirt.

[32] Qemu networking. . URL http://wiki.qemu.org/Documentation/Networking.

[33] Virtual networking. . URL https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-Virtualization_Administration_Guide-Virtual_Networking.html.

[34] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010)*, pages 166–175, Venice, Italy, 2010. IEEE Computer Society. URL http://www.sec.in.tum.de/assets/Uploads/securware2010.pdf.

[35] Leveraging forensic tools for virtual machine introspectionm. . URL https://smartech.gatech.edu/bitstream/handle/1853/38424/GT-CS-11-05.pdf. Tech.Report.

[36] Memory analysis. . URL http://www.forensicswiki.org/wiki/Memory_analysis.

[37] Linuxmemoryforensics, . URL https://code.google.com/p/volatility/wiki/LinuxMemoryForensics.