GRIZZLY: *A Framework to create signatures and launch Monitors using Virtual Machine Introspection*

FINAL REPORT

PROJECT ADVISOR: Dr. Basit Shafiq

Anas Saeed, Zirak Zaheer, Zeeshan Hakim

Department of Computer Science

SSE, LUMS

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Student Name: Anas Saeed, Zirak Zaheer and Zeeshan Hakim

Date of Submission: May 18, 2016

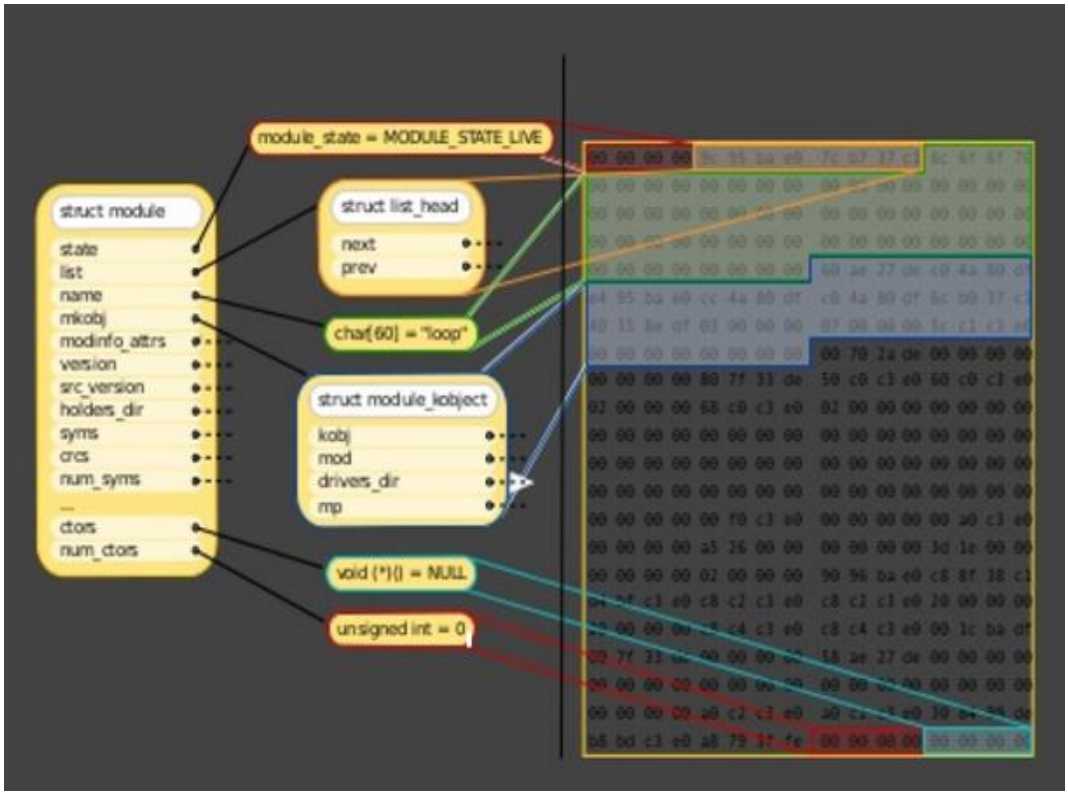
Table of Contents:

Introduction

Modern malwares are growing powerful and more sophisticated day by day. It has also been very common to see these malwares compromise the security tools along with the machine. The host based IDS tools are, therefore, not able to satisfy the security requirement of today's computer systems. Moreover, they are ineffective to detect rootkits, sophisticated keyloggers and network based attacks. The problem of host based security tools is that they run inside the same hosts they are protecting which makes them directly exposed to malware. The traditional solution to this approach has been to move the security system into network to provide isolation. But, this comes at the cost of low visibility of the OS state of the host machine. Hence, this gives attacker more room to play their tricks and compromise the system.

Drawbacks of Host based as well as Network based Intrusion Detection System has brought IDS mechanism built on Virtual Machine Introspection as a popular framework. This project aimed to design a framework using Virtual Machine Introspection to protect Virtual Machines running on a physical machine. **Virtual machine introspection** (VMI) is a technique for externally monitoring the runtime state of a system-level **virtual machine**. Monitors can be placed in another **virtual machine**, within the hypervisor, or within any other part of the virtualization architecture. Such an approach helps to shield the Intrusion Detection System even if the Virtual Machine gets compromised. At the same time it allows us to view and control the virtual machine from outside of VM. Security policy enforcement and threat detection is an essential goal of Virtual Machine introspection.

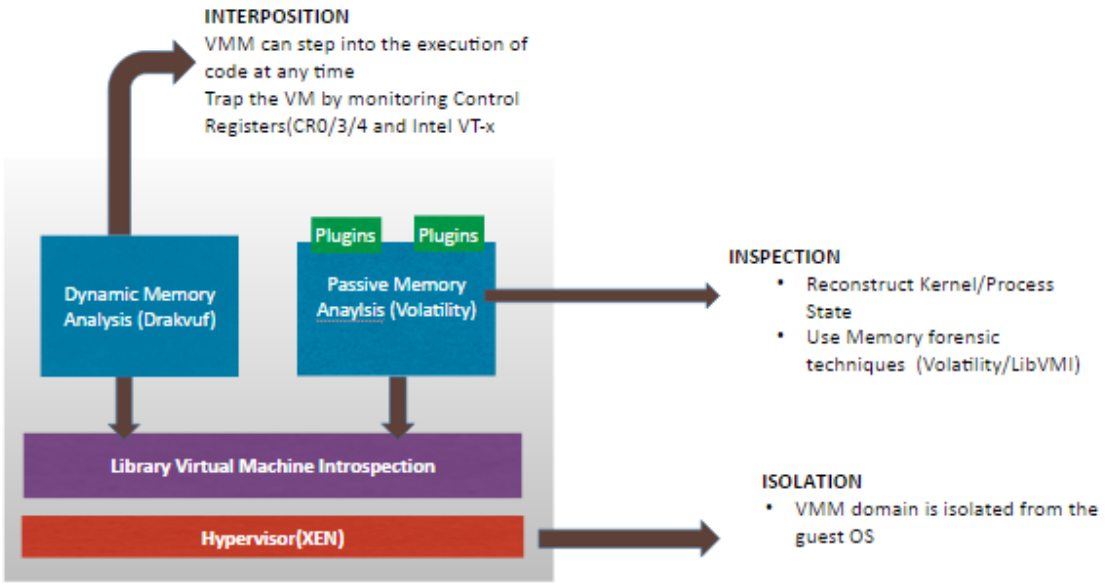
However, the problem with using VMI is of the semantic gap. Semantic gap is precisely the difference between getting between the low level memory state of the VM and the high level/meaningful OS view of the VM. The figure below depicts the problem. On the right side we have the raw memory which will be useless for the VMI framework by itself and on the left hand side we see an attempt to create some semantic meaning out of that memory dump.



Reconstructing high level state information such as knowing the processes running, sockets opened from the low level state has been made easier by memory forensics tools such as volatility and rekall.

Our framework uses XEN as the hypervisor, along with libvmi which is a library for VMI which powers the hypervisor to access and control the state of the virtual machines. For reconstructing the OS level view of the VM we are using memory forensics tool called Volatility.

The diagram below gives a high level view of the framework:



All in all the monitor sits at the hypervisor level and peeks into host using libvmi. Once state of the host has been captured we leverage volatility to track specific indicators particularly those which may go undetected by host based IDS. For example, threats that act as a lie detector are difficult to be detected without cross matching the output of multiple plugins of Volatility.

Background and Terminology

Volatility is an open source memory forensics framework for incident response and malware analysis. It is written in Python and supports Microsoft Windows, OS X, and Linux.[1] Volatility was selected because of the diverse range of plugins that it provides making it easier to reconstruct the OS level view of the memory.

Virtual Machine Monitor(VMM) is the component of the software or hardware that created and manages the virtual machines. VMM are also referred to as hypervisor.

We are using DRAKVUF for Dynamic memory analysis which allows the framework to monitor the state of the control registers and if the need be to step into the execution of the running program inside the VM at anytime. **DRAKVUF** is an agentless dynamic malware analysis system built on Xen, LibVMI, Volatility and Rekall. It allows for in-

[1] https://github.com/**volatility**foundation/**volatility**/wiki

depth execution tracing of malware samples and extracting deleted files from memory, all without having to install any special software within the virtual machine used for analysis.[2]

Virtual Machine Introspection goals may vary but in this case we focus on identifying whether a rootkit or a malicious loadable kernel etc has compromised the guest OS.

During the course of the project we have had the chance to work on two different VMI stacks. These approaches have been briefly summarized in this section:

XEN + LIBVMI

The Xen hypervisor is the basic abstraction layer of software that sits directly on the hardware below any operating systems. It is responsible for CPU scheduling and memory partitioning of the various virtual machines running on the hardware device. The hypervisor not only abstracts the hardware for the virtual machines but also controls the execution of virtual machines as they share the common processing environment [4]. It has no knowledge of networking, external storage devices, video, or any other common I/O functions found on a computing system.[3]

How does XEN work?

XEN virtual environment consists of two kinds of Virtual Machines:

1) Domain 0

It is a modified Linux kernel and is a unique virtual machine running on the Xen hypervisor that has special rights to access physical I/O resources as well as interact with other virtual machines (Domain U: PV and HVM Guests) running on the system. All Xen virtualization environments require Domain 0 to be running before any other virtual machines can be started. Domain 0 can be said to be the administrative control of the hypervisor.

[2] http://drakvuf.com/

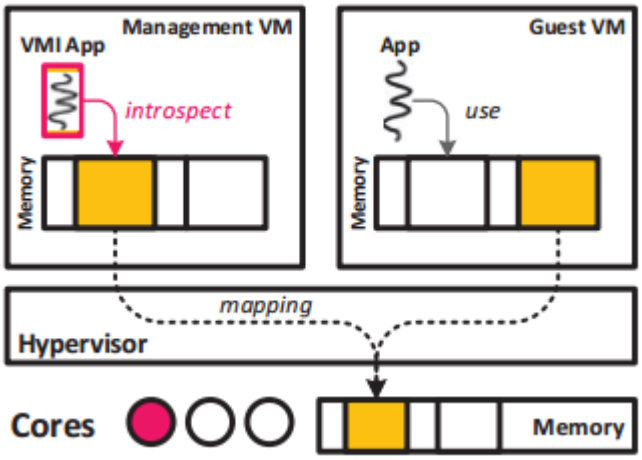
[3] http://www.fatih.edu.tr/~esma.yildirim/CENG548/HowDoesXenWork.pdf

2) Domain U

Domain U guests have no direct access to physical hardware on the machine as a Domain0 Guest does and is often referred to as unprivileged. All paravirtualized virtual machines running on a Xen hypervisor are referred to as Domain U PV Guests and are modified Linux operating systems, Solaris, FreeBSD, and other UNIX operating systems. All fully virtualized machines running on a Xen hypervisor are referred to as Domain U HVM Guests and run standard Windows or any other unchanged operating system. The Domain U PV Guest virtual machine is aware that it does not have direct access to the hardware and recognizes that other virtual machines are running on the same machine. The Domain U HVM Guest virtual machine is not aware that it is sharing processing time on the hardware and that other virtual machines are present [4].

LibVMI

LibVMI is a C library with Python bindings that makes it easy to monitor the low-level details of a running virtual machine by viewing its memory, trapping on hardware events, and accessing the vCPU registers.[4]



The figure above depicts the working of LibVMI. The VMI tool runs as a user level process inside Domian 0. Domain 0 uses the library interfaces provided by libvmi to

[4] http://libvmi.com/docs/gcode-intro.html

query the memory of the guest VM for introspection. As the VMI tool resides in a different address space than the monitored VM, it requires a two-dimensional address translation as it only gets a virtual address in the guest virtual machine. The vmi_read function gets the virtual address in the VMI tool of the specified guest virtual address. It does so by using the guest virtual address to walk the guest page table to get the corresponding guest physical address, which is then translated into host physical address and finally mapped to the virtual address in the VMI tool [3].

Due to limitations of Virtuoso and multiverse capabilities supporting VMI of XEN, we decided to build our framework on top of XEN.

Related Work:

Virtuoso

Virtuoso provides techniques to automatically create programs that can extract security-relevant information from outside the guest virtual machine. The programs are created first inside the guest OS that compute the desired information by querying the built in APIs. In contrast getting the same info from outside the guest OS requires detailed knowledge of the kernel memory and kernel structures [5].

Virtuoso uses OS's own knowledge by observing the instructions the OS performs in response to a given query. For example, in virtuoso, a program that calls getpid() is traced as it is executed. Hence, virtuoso automatically identifies the instructions necessary for running the process, and finally generated the corresponding introspection code. The program is then run inside virtuoso which records multiple executions, analyzes the resulting instruction traces and creates an out-of-guest introspection program that can retrieve the name of the process running inside a VM from outside.

Virtuoso, a system for automatically generating introspection tools that can retrieve semantically meaningful information based on low-level data sources. By applying a novel whole-system executable dynamic slicing technique, Virtuoso turns a task which once took hours or weeks of reverse engineering by an expert into one that requires

only a small amount of effort by a programmer of modest skill and a few minutes of computation time—and in doing so, helps ensure that introspection programs exactly model the behavior of the operating system. Moreover, its analysis capabilities are operating system-agnostic, removing the need for developers to constantly play catch-up as OS vendors release new versions of their products. These contributions help narrow the semantic gap, and should remove a significant roadblock in the areas of forensic analysis, virtualization-based security and low-artifact malware analysis [2].

IntroVirt

This is also VMI based project which focuses on security vulnerabilities. When a vulnerability is discovered in the system, they system administrators generally will want to run two checking phases. First they check whether the newly discovered vulnerability has attacked the system, and second they check on how they detect and respond to the vulnerability from the time of detection till the time system has been patched. IntroVirt, attempts to solve these two problems. It runs with a VMM that supports Virtual Machine replay functionality. By combining this VMM with vulnerability specific predicates it is able to detect whether some vulnerability was exploited before. Secondly, Introvirt implements some response strategies to alert or prevent the known vulnerabilities from exploitation at the hand of intruders [12].

Similar with other VMI-based systems, IntroVirt executes its predicate engine on another VM. However, the design consideration is different. In this system, the authors focus on the requirement that the vulnerability monitoring system must not perturb the target state [12]. Here, the monitoring system leverages the hardware state provided by the VMM and combines the knowledge of the target OS structure to reconstruct the system level state and finally it uses this state to monitor the vulnerable area in the target system.

Lares

Most of the VMI based security tools do passive monitoring. By passive monitoring it is meant that they only scan and poll the target system externally. Lares, however, is an active monitoring security architecture. In active monitoring the security monitor places hooks inside the monitored system. Whenever the security critical area is executed

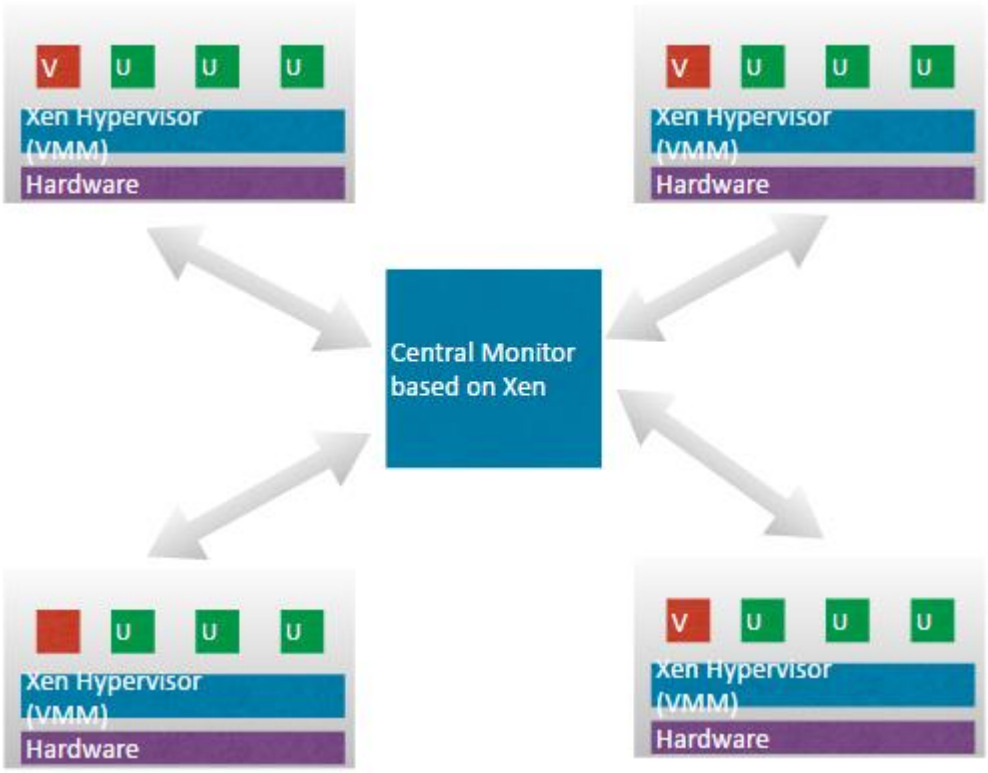
these hooks will be activated and the control will be passed to the security tool. Simply, speaking when an event of interest happens the control will be passed to the VMM via hooking. However, the switching between the guest, the hypervisor, and the secure VM is a large overhead which makes the approach extremely slow [1].

Implementation Details

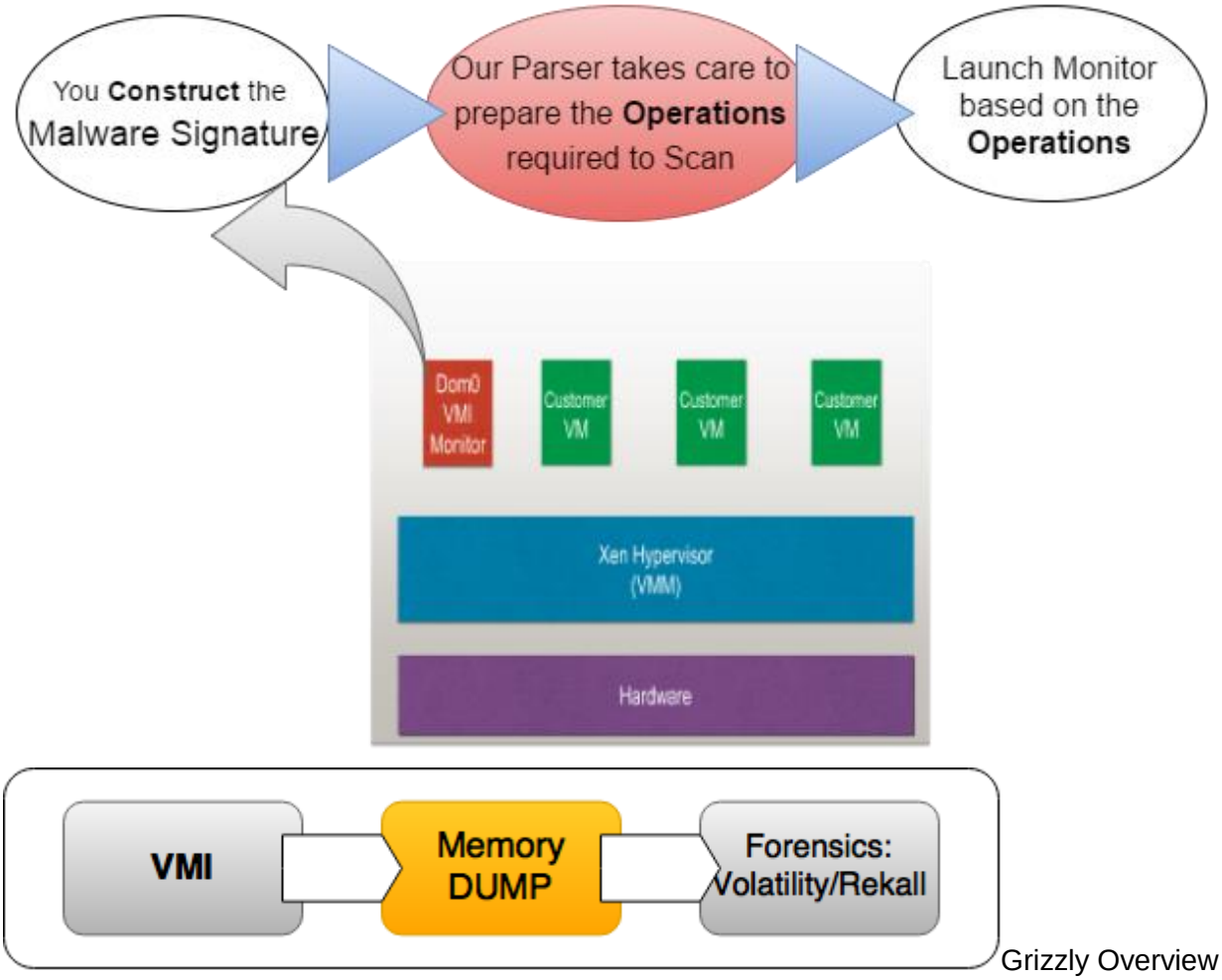
Framework Design

Grizzly is a framework that allows a user to create any malware signature and launches a monitor based on the created signature to scan the Virtual Machines in the environment. The environment currently consists of a single machine running multiple virtual machines, but this system is easily scalable to protecting multiple virtual machines across physical machines in a network setting. The high level view of the framework is depicted below.

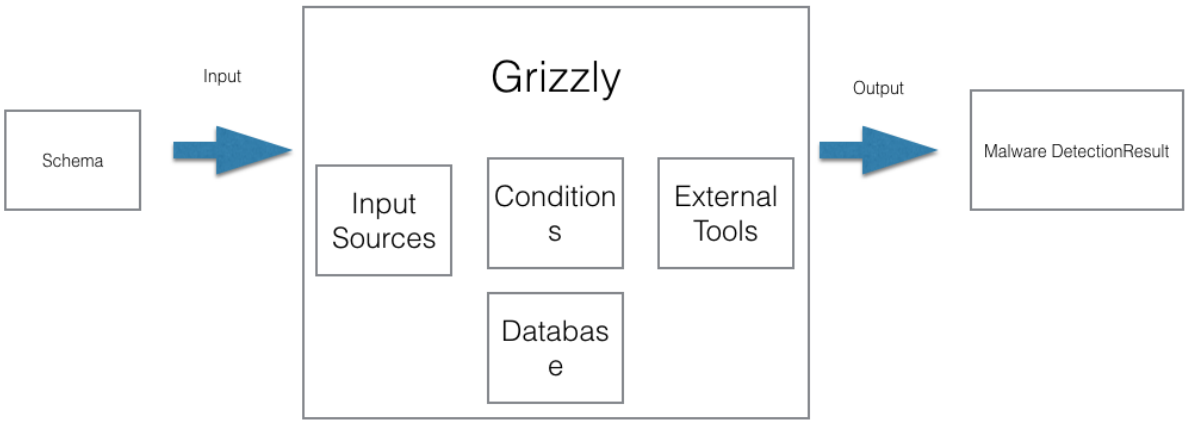
The machine in the center is responsible to log and query the virtual machine's memory. Once the monitor is launched the target virtual machines are queried at a fixed interval and monitor the changes in state of the Virtual Machines as defined in the monitors operation.



Single Machine View:



Grizzly Overview



Choice of Language

In building the system, one of the major aspect was the use of external APIs and different system calls for which native language plugins or libraries were not available like Yara or volatility. Therefore, initially bash was chosen for the system implementation as the main benefit was that it could run system commands natively unlike other languages. But, on the other hand due to the lack of high level abstractions different tasks like database integration, data structures manipulation and usage of multiple nested conditional statements became a great hassle.

Due to the above mentioned difficulties faced in bash especially after the initial phase of the project, a different language had to be chosen. Given different options which fulfilled the needs python was chosen for the various benefits that it provides, some of which are described below.

1) Open Source implementation with a plethora of tutorials and a vibrant active community.
2) Numerous libraries, extensions and plugins available leading to agile development and DRY code
3) Built in libraries for mongodb integration and xml parsing among other things
4) High level language with built in support for objection oriented programming and commonly used data structures
5) Ability to achieve more functionality with less code due to built in features and libraries.

Given the above advantages of python, it was chosen for building the system. Other languages that were also considered included C++ and Java but there lack of libraries and extensions for various tools that were needed to use for the project, they were not chosen.

Choice of Target Operating System

For experimentation of Grizzly Windows 7's 64 bit version was used. Some useful aspects of using Windows 7 are following:

1) Large installed user base

Windows 7 , released on 22 October 2009 is one of the most popular operating system for personal computers. After Windows Vista was criticised for poor performance Windows 7 came out as a great OS which while maintaining hardware and software compatibility offered great performance and earned the praise of many critics. In terms of sales, it was the highest grossing pre-order in Amazon history and the fastest selling OS in Microsoft's history. Currently Windows 7 has a market share of around 45% in the personal computer market and has sustained its share over the past few years.

2) Malware Signatures

By virtue of the large installed base of windows 7, it is lucrative target for black hat hackers. Hence there is a lot of available malware out in the open which is active and infects users. Some of the major classes of malware that are present for this version of windows are the following

a) Adware
b) Spyware
c) Keyloggers
d) Botnet Backdoors
e) Viruses and Worms

In addition to the classes, some specific malwares that are active on this platform are the following

a) Taterf
b) Renos
c) Alureon
d) FakeRean
e) Bancos

3) Virtualization Support

In addition to the large user base and malware samples that are available for the platform, Windows 7 can easily be virtualized in the environment we setup which used Xen and Debian. The virtualization process is not only straightforward, but also windows 7 runs smoothly and the overhead along with the restriction of functionality due to virtualization are low.

Although for testing and experimentation purposes , Windows 7's 64 bit version was used. The system is universal and not at all limited to any one OS version or any one OS for that matter.

Choice of VMI OS

For virtualization of a VM we used Xen as a hypervisor and Debian as the Domain 0 (which is required as part of the Xen virtualization architecture). Some of the benefits of using Xen were the following:

1) Performance Boost:

Xen runs most instructions directly on the hardware without any emulation, the number of traps and are reduced to bare minimum and exception are handled efficiently through an already installed exception handler (that the guest OS provides).

2) Porting Effort:

In Xen porting a OS to run on a VM does not require any changes to the OS and all the complexity is abstracted away. Hence while using Xen we could test different OS without significant effort.

3) Configuration:

Setting up Xen and doing all the required configuration for booting a VM instance is quite straightforward and the plethora of online tutorials and help available makes debugging an issue quite simple. Hence, while using Xen we could set up system within a day and get started with the testing.

4) VM Setup:

One main benefit of using Xen as a hypervisor is that booting up a VM is seamless and various parameters e.g. memory can be tuned for each VM which is a great help in the testing phase.

All the setup and testing of Grizzly was done in Debian and the reasons are the following:

1) Open Source and Freely available Operating System
2) An active community with great online support in the form of tutorials and different forums.
3) Debian is very versatile with support for many different languages and tools that were utilized for Grizzly.

Usage

Installation

Grizzly is an automated malware detection framework that is based on virtual machine introspection for malware detection. For installing Grizzly the following steps need to be taken.

1) Install Python

2) Install Mongodb
3) Install Volatility advanced memory forensic analysis tools
4) Install Rekall memory forensics tools
5) Install Yara

These were all the dependencies of Grizzly.

Next get the code from github and place it in an appropriate folder for usage.

Invocation

For using Grizzly the main file MultipleOperations.py has to be called that is in the Parser folder of Grizzly's codebase. As with other python files, by using terminal and python command we can run the file.

The following command is used start a monitor for the given schema

```
python MultipleOperations.py Schema.xml
```

Here MultipleOperations.py is the main file of Grizzly and runs the schema, on the other hand Schema.xml is a command line argument whcih tells Grizzly which schema to run.

If the invocation is from outside Grizzly's folder then the following command is used.

```
python /Path_To_Grizzly/Parser/MultipleOperations.py Schema.xml
```

If the schema is in another folder from which the invocation is happening then the following command is used

```
python MultipleOperations.py /Path_To_Schema/Schema.xml
```

In some schemas due to the nature of primitives involved, root access may be required.

Input tools

The tools used for information extraction are the following:

1) Volatility

Volatility is an advanced memory forensics analysis tool which is written in Python and available under GNU General Public License. It is used for extraction of digital artifacts like process or file data from RAM dump. Volatility can extract useful

information over 50 different operating system dumps including Windows, Mac, Linux and both Android and ARM processor, some of the prominent ones are as follows:

1) 32 and 64 bit Windows 7
2) 32 and 64 bit Windows XP
3) 32 and 64 bit Windows 8
4) 64 bit Windows 10
5) 32 and 64 bit Linux Kernels from 2.6.11 too 4.2.3
6) Different Linux flavors like OpenSuSe, Ubuntu Debian, CentOS and Fedora
7) 64 bit version of Mountain Lion, Mavericks. Yosemite and El Capitan
8) 32 and 64 bit version of Snow Leopard

Volatility gives detailed information in various formats and details about different parts of the system based on the RAM data which is fed into the volatility framework. Some of the main areas of concern for or project regarding volatility are the following:

1) Processes
2) Networking
3) Threads
4) Registry
5) Process Memory

For Grizzly each of these domains are important and Volatility is used to gather information in these domains using the various commands and plugins of the framework, The data is then cleansed, parsed, stored and processed by Grizzly.

Volatility is also a well designed framework with detailed documentation and multiple available extensions which are made by its community and this aspect of the framework is very beneficial for Grizzly as any type of modification or add on is possible.

2) Rekall

Rekall similar to Volatility is a memory forensics tools and is used in Grizzly for the collection of data from memory samples which is then processed [7]. Rekall is an end to end memory forensics tools that only takes the memory dump as input and based on the command or plugin being used outputs the relevant data which is used by Grizzly. Some salient features of Rekall relevant to Grizzly are the following:

1) Rekall is an open source project which supports memory forensics for major platforms.
2) A major benefit of using Rekall, is that it is easily integrated into the system as it can be used as a library in Grizzly.

3) On the other hand its output can be formulated in various ways which reduces the post data gathering cleansing process.
4) Rekall utilizes symbols that are obtained from operating system vendors' debugging information for the processing, through which it can know the position of critical OS constants. This is one of the differentiating factor of Rekall from other memory forensics tools.

Similar to the usage of Volatility in Grizzly, Rekall is also used for gathering data from the memory dumps of a VM that is to be tested. The data is passed to different phases which include cleansing, parsing, storing and then usage for querying.

The different domains of data for which Rekall is used are the following:

1) Processes
2) Networking
3) Threads
4) Files
5) Process Memory

Although some domains overall with Volatility, the information gleaned from both the tools are often in different aspects and details due to which both tools have to be used and supplement each other for information gathering.

Documentation

The basic operations that can be used for defining a schema along with their detail are the following:

Alert:

By using the alert option, the user will be given a clear message upon the event happening along with the value and level the schema has defined.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="Alert" />
```

```
        <operationInputFile myvalue="Value of Alert" />
        <operationPath myvalue="Alpha" />
    </operation>
</operations>
```

In the above example, when alert is generated, its value will be “Value of Alert Alpha” and level will be ALpha. Its level is introduced,so that only relevant data is shown to the users. Levels is a tunable parameter and if the value is true then the alert will be generated, else it will not be hence the users can remove the clutter and only view relevant data.

Custom File:

While working with Grizzly sometimes we needed to use custom functionality e.g. aggregate the data with a particular condition etc which could not have been generalized for baking into the framework. Hence the solutions that we used was give the user the ability to write his own custom code and have Grizzly execute it.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="CustomPython" />
        <operationInputFile myvalue="custom.py" />
        <operationPath myvalue="Custom" />
    </operation>
</operations>
```

This code will instruct Grizzly to execute custom.py python file which is located in the Custom folder. Please note that only python files can be executed, but the framework can easily be extended to other languages. We did not implement to usage of other languages as our whole framework was in python and it's a very versatile language.

If Condition

This operation is essential to our framework and is used to check conditions based on the values that are already stored in the database.

Syntax:

Simple If Condition

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="ifMultiple" />
		<ifConditionMultiple myvalue="or">
			<ifConditionBasic>
				<key myvalue="keyValue" />
				<value myvalue="rvalue" />
				<collection myvalue = “collectionValue”/>
				<operator myvalue="equals" />
			</ifConditionBasic>
		</ifConditionMultiple>
	</operation>
</operations>
```

The syntax for a basic if condition is above. It only checks for one condition and please note that the If Condition Multiple myvalue is or, this is redundant and the and operator can also be used. The operator, checks for which type of checking is to be done. There are multiple operators available in Grizzly and their details are the below. And similarly , the key field what to check against, the value field contains the value we want to check for and the collection is the database table in which we want to check.

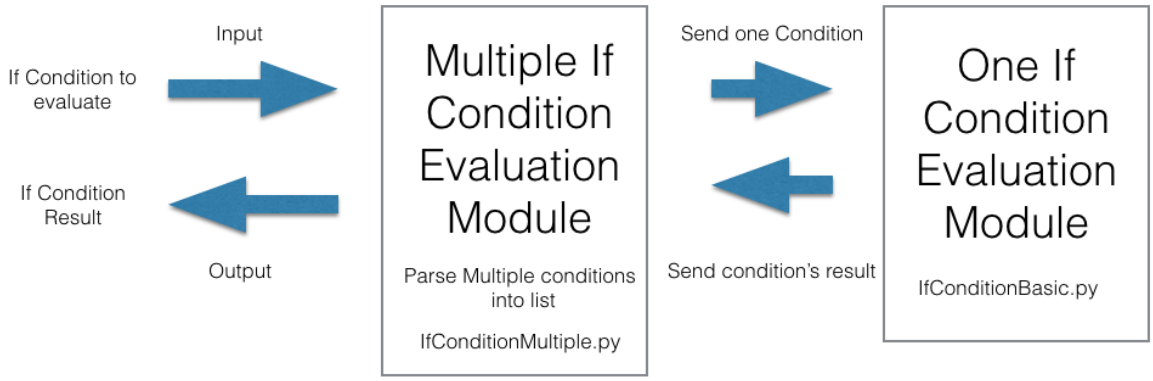
Multiple If Conditions

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="ifMultiple" />
		<ifConditionBasic>
				<key myvalue="keyValue" />
				<value myvalue="value" />
				<collection myvalue = “collectionValue”/>
				<operator myvalue="equals" />
			</ifConditionBasic>
			<ifConditionBasic>
				<key myvalue="keyValue" />
```

```
                    <value myvalue="value" />
                    <collection myvalue = “collectionValue”/>
                    <operator myvalue="equals" />
                </ifConditionBasic>

            </ifConditionMultiple>
        </operation>
</operations>
```

This is the basic syntax for a multiple if condition. The if conditions can be grouped together and based upon the If Condition Multiple value either “and” or “or” operator will be used to evaluate the results of the conditional statements and their result is then conveyed to the user.



If Condition Evaluation mechanism

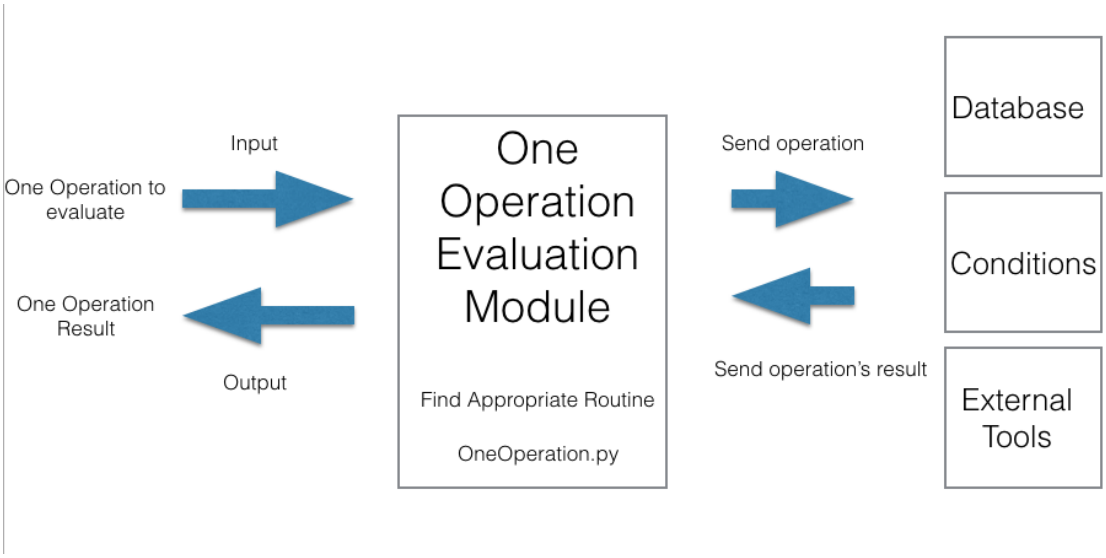
One Operations:

In Grizzly all the basic primitives, if conditions and other actions are modeled as operations and they are written and executed as one operation each for simplicity bases.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="cybox" />
         // different parameters
    </operation>
</operations>
```

All one operation are included in multiple operations and they are executed in this manner.



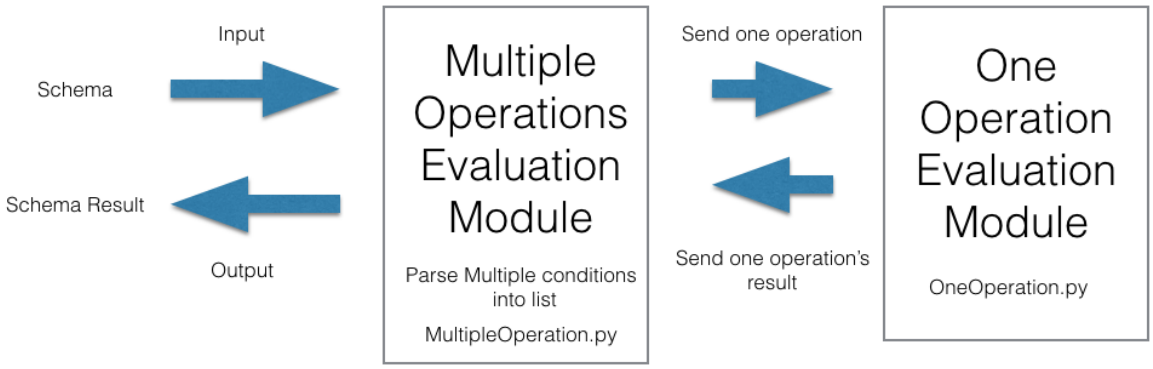
Multiple Operations

In Grizzly, all the primitives are simple operations and hence they are grouped under multiple operations, which is the outermost tag. The list of operations included in multiple operations are then executed in a linear manner.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    // all the different operations to be used
</operations>
```

This is the code for multiple operations, the different operations that are to be executed are written in between the operations tag and Grizzly executes them one by one and one at a time.



Evaluation Mechanism in Grizzly

Nested Schemas

For running schemas from within schemas, nesting has been implemented and the level of recursion is not limited in any manner. This was done for incorporating complex schemas for which modularity and separations of related material are important to make the code readable and maintainable.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="nestedOperations" />
        <nestedOperation myvalue="path to nested Schema file" />
    </operation>
</operations>
```

Here the path to the nested Schema file which is basically the name of the file along with the folder (if any) in which it preset. Normally a separate folder is made for keeping all of the nested schemas together. The benefit of this approach is that the level of recursion is infinite and there is no difference between the definition of nested and normal schemas. Any normal schema can be used as a nested schema.

Cybox

While researching for Grizzly and implementing it,we found another language called Cybox which was quite useful as it already had the parser, documentation and generator implemented. Hence, instead of making the wheel again we used Cybox for many things that it had already implemented [8].

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
        <operation>
                <operationName myvalue="cybox" />
                <cyboxXml myvalue="path to file should be given here" />
        </operation>
</operations>
```

Similar to the nested schemas, the path which is the name along with the folder in which the file is present is given to the schema and Grizzly executes it. Here one important thing is that since the language and interpreter of Cybos is different, when a cybox operation comes. The Grizzly parser gives control to a separate function which then does all the cybox executing and returns control. This proxy based design pattern allows us to abstract away cybos complexity from our code and simplify it.

Get Dump:

In Grizzly, all the operations are executed on the volatile memory data and it forms the basis of all the information gathering and knowledge extraction. For obtaining the dump of any running VM the get dump function is used.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
        <operation>
                <operationName myvalue="getDump" />
                <vmName myvalue="Virtual machine Name" />
                <operationPath myvlaue  = “dolder which includes the file”/>
                <operationInputFile myvalue= “ file to write data”/>
        </operation>
```

```
</operations>
```

Here when the above command is executed, Grizzly takes a dump of the virtual machine specified by calling one of its routines and saves the output to the file. Taking dump of a virtual machine is quite time taking task as to amount of data to be written is in GBs, hence dumps should be taken frequently but rather after fixed intervals.

Exit

By using this command, the system's executing can be terminated at any given point. It is mainly used to terminate the program if the purpose of the schema is fulfilled and there is no need to run all of it as it has detected the threat.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="exit" />
	</operation>
</operations>
```

Bulk Extractor

Bulk_extractor is one of the many tools that we use in the system. It basically takes a RAM dump and generates a pcap file which includes all the data related to network activity including the IP address, timestamps and other data related to packets. The pcap file is also very handy in further processing of network data.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="Bulk_Extractor" />
		<operationInputFile myvalue="MemoryDumps/DOS.vmem" />
		<operationPath myvalue="MemoryDumps/Bulk_Extractor" />
	</operation>

</operations>
```

TCP DUMP

This tool is also used for network related activity and its input is the pcap file which can be generated from Bulk extractor. It outputs all the packets and their related data, which is in such a manner that it can easily be passed through the cleansing, parsing and storing phase.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="TCP_DUMP" />
		<operationInputFile myvalue="MemoryDumps/Bulk_Extractor/packets.pcap" />
	</operation>
</operations>
```

If Condition Operators

In Grizzly there are multiple operators available for conditional checking Grizzly.

1) Contains

Checks wether that the value exists in the collection under the for the field key.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
	<operation>
		<operationName myvalue="ifMultiple" />
		<ifConditionMultiple myvalue="or">
			<ifConditionBasic>
				<key myvalue="keyValue" />
				<value myvalue="value" />
				<collection myvalue = “collectionValue”/>
				<operator myvalue="contians" />
			</ifConditionBasic>
		</ifConditionMultiple>
	</operation>
</operations>
```

2) Greater Than

Check whether the value of key in the database is greater than the value (Specified in schema) or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="ifMultiple" />
        <ifConditionMultiple myvalue="or">
            <ifConditionBasic>
                <key myvalue="keyValue" />
                <value myvalue="value" />
                <collection myvalue = “collectionValue”/>
                <operator myvalue="greaterThan" />
            </ifConditionBasic>
        </ifConditionMultiple>
    </operation>
</operations>
```

3) Top Sender Sum

This operator is used to check if the sum of top n of the key (which is a database field) is greater than the value or not. If first sorts the keys then checks selects the top n and sees if their sum of values are greater than the required value or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="ifMultiple" />
        <ifConditionMultiple myvalue="or">
            <ifConditionBasic>
                <key myvalue="keyValue" />
                <value myvalue="value" />
                <collection myvalue = “collectionValue”/>
                <operator myvalue="topSenderSum" />
```

```
                </ifConditionBasic>
            </ifConditionMultiple>
        </operation>
</operations>
```

4) Top Senders

This operator is used to check if the sum of all the key (which is a database field) is greater than the value or not. If first sorts the keys then checks selects the top n and sees if each of its values are greater than the required value or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
        <operation>
                <operationName myvalue="ifMultiple" />
                <ifConditionMultiple myvalue="or">
                        <ifConditionBasic>
                                <key myvalue="keyValue" />
                                <value myvalue="rvalue" />
                                <collection myvalue = “collectionValue”/>
                                <operator myvalue="topSenders" />
                        </ifConditionBasic>
                </ifConditionMultiple>
        </operation>
</operations>
```

5) Average

This operator is used to check if the average of all the key (which is a database field) is greater than the value or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
        <operation>
                <operationName myvalue="ifMultiple" />
                <ifConditionMultiple myvalue="or">
```

```
                    <ifConditionBasic>
                        <key myvalue="keyValue" />
                        <value myvalue="rvalue" />
                        <collection myvalue = “collectionValue”/>
                        <operator myvalue="average" />
                    </ifConditionBasic>
                </ifConditionMultiple>
            </operation>
        </operations>
```

6) Sum

This operator is used to check if the sum of all the key (which is a database field) is greater than the value or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
    <operation>
        <operationName myvalue="ifMultiple" />
        <ifConditionMultiple myvalue="or">
            <ifConditionBasic>
                <key myvalue="keyValue" />
                <value myvalue="rvalue" />
                <collection myvalue = “collectionValue”/>
                <operator myvalue="sum" />
            </ifConditionBasic>
        </ifConditionMultiple>
    </operation>
</operations>
```

7) Equals

This operator is used to check if the key and value are equal or not.

Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<operations>
```

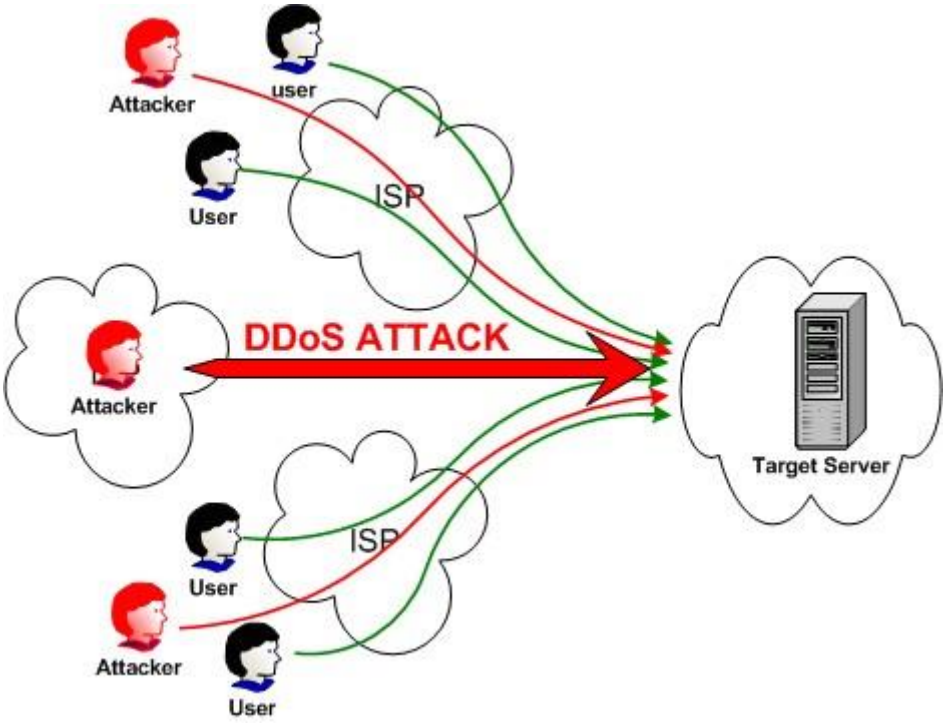
```
        <operation>
            <operationName myvalue="ifMultiple" />
            <ifConditionMultiple myvalue="or">
                <ifConditionBasic>
                    <key myvalue="keyValue" />
                    <value myvalue="value" />
                    <collection myvalue = “collectionValue”/>
                    <operator myvalue="contians" />
                </ifConditionBasic>
            </ifConditionMultiple>
        </operation>
    </operations>
```

Attack Detection:

The categories of attacks on which Grizzly is effective are the following:

1) Network

Denial of service attacks is an attempt to take the target system offline by swamping it with network traffic to a limit that it can not cope with. The use of botnets, attack amplification and other sophisticated techniques has made DDOS a deadly attack for servers that are active on the internet. Attacks have been peaking at more that 1,000 Gbps and 50 million packets per second with an increase of more than 17% in 2014. Given the ubiquity of the internet such attacks are ever present and increasing becoming more and more deadly.



For testing a virtual machine instance for DOS we initially took the dump of the system and then ran bulk_extractor on the dump which gives us the pcap file. This pcap file includes the detailed information about the packets in that have traversed the system, their source and destination IP address in a raw form. By using the pcap file which is generated from bulk extractor we feed it into another tool TCPDump which makes the output more refined and gives us a clean picture of the network in the form of packet tuples. Once the packet information is gathered it is stored into the database of querying.

After storing the data, by using the operators provided by Grizzly we can apply different conditions for detecting whether our system was attacked or no. The operators that we used are the following

1) Contains

We check if the number of packets from a particular IP are more than a given threshold which can be dynamically set.

2) Sum

This operator checks of the total number of packets in a time frame are greater than a threshold that the user procides.

3) Average

Average operator finds the average of packets over a time frame and the condition checks for the threshold.

4) Top Senders

In this operator, we first aggregated the IPs and then sort them on the basis of the number of packets. From these we select the the top N (user provided number) IPs and check whether the number is greater a threshold or no.

5) Top Senders Sum

This operator is similar to top senders, the only difference is that instead of checking for each IP we sum them and check whether their packets sum is greater then the user provided threshold or no.

Code

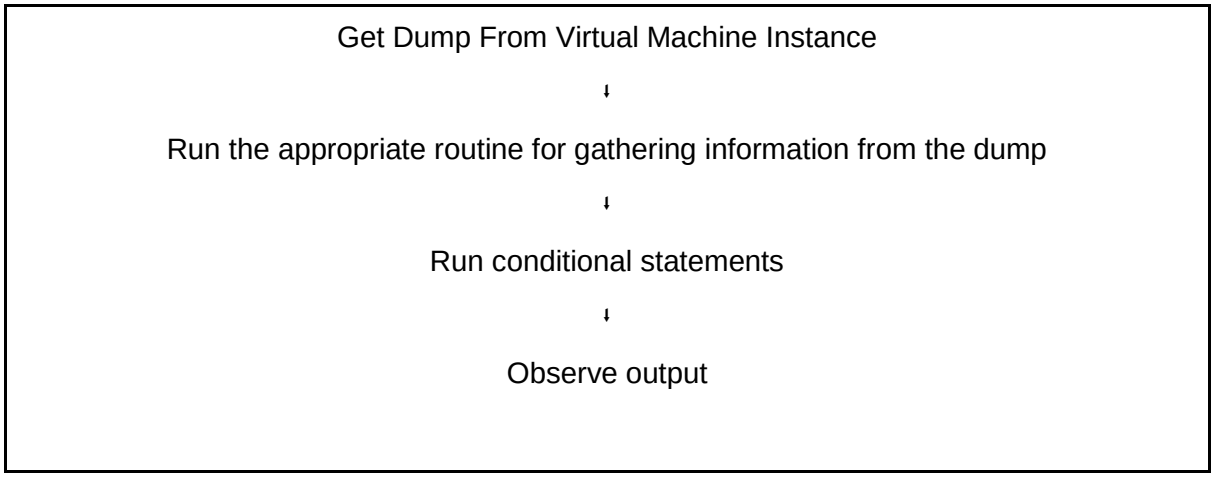
```
<?xml version="1.0" encoding="UTF-8"?>
<operations description="DOS attack detection script">
	<operation>
		<operationName myvalue="Bulk_Extractor" />
		<operationInputFile myvalue="MemoryDumps/DOS.vmem" />
		<operationPath myvalue="MemoryDumps/Bulk_Extractor" />
	</operation>
	<operation>
		<operationName myvalue="TCP_DUMP" />
		<operationInputFile myvalue="MemoryDumps/Bulk_Extractor/packets.pcap" />
	</operation>
	<operation>
		<operationName myvalue="ifMultiple" />
		<ifConditionMultiple myvalue="and">
			<ifConditionBasic>
				<key myvalue="256.221.251.219" />
				<value myvalue="5000" />
				<collection myvalue="DOSCollection" />
				<operator myvalue="greaterThan" />
			</ifConditionBasic>
```

```
        <ifConditionBasic>
                <key myvalue="5000" />
                <value myvalue="5" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="topSendersSum" />
        </ifConditionBasic>
        <ifConditionBasic>
                <key myvalue="10" />
                <value myvalue="5" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="topSenders" />
        </ifConditionBasic>
        <ifConditionBasic>
                <key myvalue="168.97.85.92" />
                <value myvalue="5000" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="contains" />
        </ifConditionBasic>
        <ifConditionBasic>
                <key myvalue="56.21.51.19" />
                <value myvalue="5000" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="contains" />
        </ifConditionBasic>
        <ifConditionBasic>
                <key myvalue="" />
                <value myvalue="1000" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="average" />
        </ifConditionBasic>
        <ifConditionBasic>
                <key myvalue="" />
                <value myvalue="5000" />
                <collection myvalue="DOSCollection" />
                <operator myvalue="sum" />
        </ifConditionBasic>
</ifConditionMultiple>
```

```
        </operation>
</operations>
```

2) Lie Detector

In security, checking for items that should not be present is essential and native to each anti virus. We also implement such functionality but with power of VMI. The normal process for detecting whether any black listed item is present or not is the following:



The BlackLists were implemented for the following domains

1) IP Addresses
2) Processes
3) DLLs
4) Commands
5) Files
6) APIs
7) Processes using network

3) Rootkits

Rootkits are malicious and stealthy pieces of software that are designed to hide their existence using evasive measures for ensuring they are not detected using normal mechanisms and simultaneously continue to execute on the machine at privileged level. Due to the escalated privileges that the rootkit has it can do things that normal malware can not do and hence it is more deadly. One of the main mechanisms that rootkits deploy for subverting detection mechanisms, is to remove itself from the process list that the operating system maintains which is a double linked list. Hence, the rootkit is not visible in the task manager of windows and a

normal antivirus cannot detect it. Another aspect of rootkit is that it sometimes loaded before windows and hence it becomes completely invisible to the operating system.

To counter all of these subversion mechanisms that black hat people deploy we come up with different ways of detecting the rootkits which are the following:

1) Using Process List:

Initially we just look at the operating system process list and walk the double link list to find if there is a rootkit present in the system or not. By using PsList routine of Grizzly we can obtain the process list of the operating system.

2) Ready queue

Whenever the system has to run any process it places it in the ready queue and hence even if the rootkit removes it from the process list, it has to come in the ready queue if it has to be executed. Hence by continuously monitoring this crucial data structure. We can detect the root kits that active in the system. By using PsTree commands we can get the processes in the ready queue and then by using if conditions we were able to obtain the required results.

3) Scanning Tags

Third we also enumerated the pool tag (_POOL_HEADER) and DISPATCHER_HEADER for finding all the processes in the memory. Using these methods we can find previously terminated process that have been hidden or unlinked by the rootkit. Psscan and Psdispscan are the commands for enumerating the tags.

Code

```
<?xml version="1.0" encoding="UTF-8"?>
<operations description="Script for checking the presence of a rootkit, by checking its presence in three different databases">
	<operation>
		<operationName myvalue="PsList" />
		<operationInputFile myvalue="MemoryDumps/Rootkit.vmem" />
		<operationPath myvalue="Primitives/Proceses/PsList" />
	</operation>
	<operation>
		<operationName myvalue="Psscan" />
```

```
        <operationInputFile myvalue="MemoryDumps/Rootkit.vmem" />
        <operationPath myvalue="Primitives/Proceses/Psscan" />
</operation>
<operation>
        <operationName myvalue="PsxView" />
        <operationInputFile myvalue="MemoryDumps/Rootkit.vmem" />
        <operationPath myvalue="Primitives/Proceses/PsxView" />
</operation>
<operation>
        <operationName myvalue="ifMultiple" />
        <ifConditionMultiple myvalue="and">
                <ifConditionBasic>
                        <key myvalue="PsList-Name" />
                        <value myvalue="agpbrdg5.sys" />
                        <collection myvalue="PsListCollection" />
                        <operator myvalue="contains" />
                </ifConditionBasic>
                <ifConditionBasic>
                        <key myvalue="PsList-Name" />
                        <value myvalue="alcop.sys" />
                        <collection myvalue="PsListCollection" />
                        <operator myvalue="contains" />
                </ifConditionBasic>
        </ifConditionMultiple>
</operation>
<operation>
        <operationName myvalue="ifMultiple" />
        <ifConditionMultiple myvalue="and">
                <ifConditionBasic>
                        <key myvalue="Psscan-Name" />
                        <value myvalue="armdvc.sys" />
                        <collection myvalue="PsscanCollection" />
                        <operator myvalue="contains" />
                </ifConditionBasic>
                <ifConditionBasic>
                        <key myvalue="Psscan-Name" />
                        <value myvalue="63cica.sys" />
```

```
                        <collection myvalue="PsscanCollection" />
                        <operator myvalue="contains" />
                    </ifConditionBasic>
                </ifConditionMultiple>
            </operation>
            <operation>
                <operationName myvalue="ifMultiple" />
                <ifConditionMultiple myvalue="and">
                    <ifConditionBasic>
                        <key myvalue="PsxView-Name" />
                        <value myvalue="agehhtd.cat" />
                        <collection myvalue="PsxviewCollection" />
                        <operator myvalue="contains" />
                    </ifConditionBasic>
                    <ifConditionBasic>
                        <key myvalue="PsxView-Name" />
                        <value myvalue="aiqpbter.chm" />
                        <collection myvalue="PsxviewCollection" />
                        <operator myvalue="contains" />
                    </ifConditionBasic>
                </ifConditionMultiple>
            </operation>


             <operation>
  <operationName myvalue="DllList" />
  <operationInputFile myvalue="MemoryDumps/Rootkit.vmem" />
  <operationPath myvalue="Primitives/Proceses/DllList" />
</operation>

<operation>
  <operationName myvalue="ifMultiple" />
  <ifConditionMultiple myvalue="and">
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\ntshrui.dll" />
      <collection myvalue="DllListCollection" />
```

```
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\OLEAUT32.dll" />
      <collection myvalue="DllListCollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\bpki.dll" />
      <collection myvalue="DllListCollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
  </ifConditionMultiple>
</operation>
<operation>
            <operationName myvalue="ifMultiple" />
            <ifConditionMultiple myvalue="and">
                  <ifConditionBasic>
                        <innerCollection myvalue="PsListCollection" />
                        <outerCollection myvalue="PsxViewCollection" />
                        <innerCollectionField myvalue="PsList-Name" />
                        <outerCollectionField myvalue="PsxView-Name" />
                        <operator myvalue="Inversion" />
                  </ifConditionBasic>
            </ifConditionMultiple>
      </operation>
      <operation>
            <operationName myvalue="ifMultiple" />
            <ifConditionMultiple myvalue="and">
                  <ifConditionBasic>
                        <innerCollection myvalue="PsListCollection" />
                        <outerCollection myvalue="PsscanCollection" />
                        <innerCollectionField myvalue="PsList-Name" />
                        <outerCollectionField myvalue="Psscan-Name" />
                        <operator myvalue="Inversion" />
```

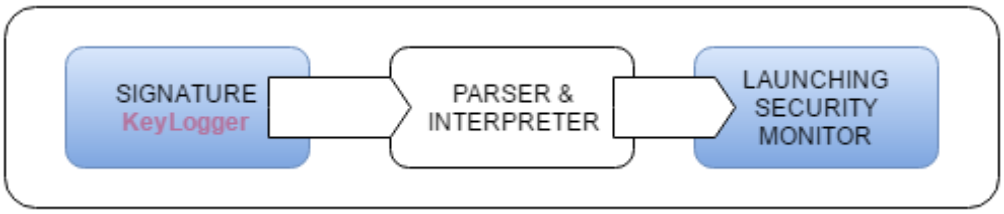
```
                    </ifConditionBasic>
                </ifConditionMultiple>
            </operation>

</operations>
```

4) Keylogger

Of the many types of malware out there in the open, key logger is one of the worst as it intercept all the passwords, banking details and email among others. There are mainly three categories of key loggers.

1) Hardware based
2) User Mode
3) Kernel Mode



In user mode the most common technique is to use Windows API SetWindowsHookex for intercepting the system events and then capturing the data. ON the other hand kernel mode keyloggers are more difficult to detect and use code injection, dynamically loaded libraries and filter drivers for intercepting the data.

Hardware keyloggers are devices that are placed between the keyboard and CPU so that they can store all the data that is transferred on the wire.

For detecting keyloggers we use multiple mechanisms as the black hat people use multiple ways to subvert the system. The mechanisms are the following

1) Blacklisting

In this part we simply get a list of all the processes running on the system from the process list, ready queue and other sources and check whether there exists a process which should not. For this we use the PsList and Psscan routines and use if conditions for checking the presence of any mischievous process.

2) Connection State

Keyloggers normally collect data on a remote machine and then transfer that to a remote location, so in this part of the schema we check if there is an external connection for a process that should not be present. For detecting connection state we use Connscan and then compare them.

3) Dynamically Loaded Libraries (DLLs)

Some key loggers get installed as normal programs and do not any malicious activity but instead they load a DLL and all the dirty work is offloaded to it and this is exactly what we check for. By using Dll List command we can find a list of all the DLLs used and then compare them using the if conditions.

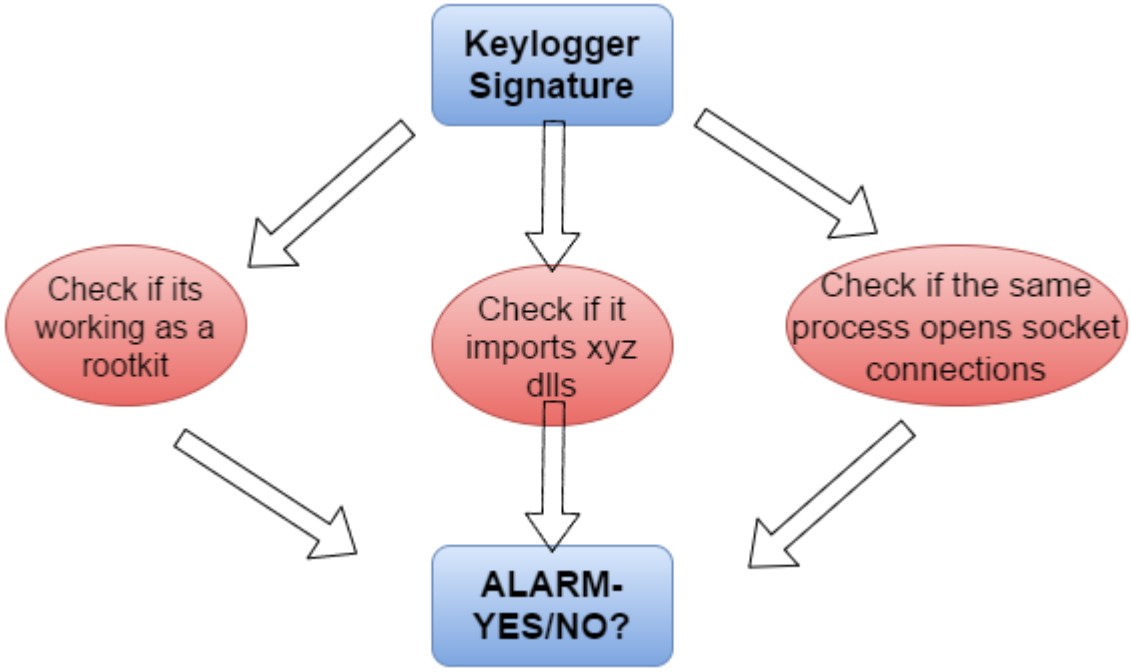
4) WIndows API

As mentioned before, user mode key loggers mostly run using windows API and intercept the system events and then the keyboard data. Such key loggers are easy to implement and hence are the most widely used. By obtaining the list of used APIs using the WindowsAPI routine of Grizzly we can cross check it for detecting malicious activity.

5) Code Injection

Mostly keyloggers come in the system as harmless programs and once installed get to do all the dirty work they are designed for. Code injection is one of the eminent example of this behaviour, but by using an external tool -Yara, Grizzly is able to detect code injection along with other malicious signatures using the tool.

----------High Level View----------



Code

```
<?xml version="1.0" encoding="UTF-8"?>
<operations description="Script for Detecting Key Logger using Name, Connections, Dlls Code Injection and Windows APIs">
  <operation>
    <operationName myvalue="PsList" />
    <operationInputFile myvalue="MemoryDumps/KeyLogger.vmem" />
    <operationPath myvalue="Primitives/Proceses/PsList" />
  </operation>
  <operation>
    <operationName myvalue="ifMultiple" />
    <ifConditionMultiple myvalue="or">
      <ifConditionBasic myvalue="Process object">
        <key myvalue="PsList-Name" />
        <value myvalue="keylogger.exe" />
        <collection myvalue="PsListCollection" />
        <operator myvalue="contains" />
      </ifConditionBasic>
      <ifConditionBasic myvalue="Process object">
        <key myvalue="PsList-Name" />
        <value myvalue="keyloggerxyz.exe" />
        <collection myvalue="PsListCollection" />
```

```
      <operator myvalue="contains" />
    </ifConditionBasic>
  </ifConditionMultiple>
</operation>
<operation>
  <operationName myvalue="Connscan" />
  <operationInputFile myvalue="MemoryDumps/KeyLogger.vmem" />
  <operationPath myvalue="Primitives/Networking/Connscan" />
</operation>
<operation>
  <operationName myvalue="ifMultiple" />
  <ifConditionMultiple myvalue="or">
    <ifConditionBasic>
      <key myvalue="Connscan-Remote-Address" />
      <value myvalue="192.154.6.7" />
      <collection myvalue="testcollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
      <key myvalue="Connscan-Remote-Address" />
      <value myvalue="192.154.6.88" />
      <collection myvalue="testcollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
  </ifConditionMultiple>
</operation>
<operation>
  <operationName myvalue="DllList" />
  <operationInputFile myvalue="MemoryDumps/KeyLogger.vmem" />
  <operationPath myvalue="Primitives/Proceses/DllList" />
</operation>
<operation>
  <operationName myvalue="ifMultiple" />
  <ifConditionMultiple myvalue="and">
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\ntshrui.dll" />
```

```
      <collection myvalue="DllListCollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\OLEAUT32.dll" />
      <collection myvalue="DllListCollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
      <key myvalue="DllList-Path" />
      <value myvalue="C:\WINDOWS\system32\bpki.dll" />
      <collection myvalue="DllListCollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
  </ifConditionMultiple>
</operation>
<operation>
  <operationName myvalue="WindowsAPI" />
  <operationInputFile myvalue="MemoryDumps/KeyLogger.vmem" />
  <operationPath myvalue="Primitives/Proceses/WindowsAPI" />
</operation>
<operation>
  <operationName myvalue="ifMultiple" />
  <ifConditionMultiple myvalue="and">
    <ifConditionBasic>
      <key myvalue="WindowsAPI-Name" />
      <value myvalue="SetWindowsHookEx" />
      <collection myvalue="WindowsAPICollection" />
      <operator myvalue="contains" />
    </ifConditionBasic>
    <ifConditionBasic>
    <key myvalue="WindowsAPI-Name" />
    <value myvalue="GetAsyncKeyState" />
    <collection myvalue="WindowsAPICollection" />
    <operator myvalue="contains" />
  </ifConditionBasic>
```

```
    </ifConditionMultiple>
  </operation>
  <operation>
    <operationName myvalue="Yara" />
    <operationInputFile myvalue="MemoryDumps/KeyLogger.vmem" />
    <operationPath myvalue="Primitives/Yara" />
  </operation>
</operations>
```

Limitations

The main limitations of Grizzly are the following:

1) Semantic Gap

One of the main issues with VMI is creating the high level information which can be used from low level ones and zeros that are available. This is one of the main issues that we also faced during the design and implementation of Grizzly as it severely limited the useful information that was available. Creating useful information from the bits available through the dump is very cumbersome as reverse engineering the operating system, its data structures and its routines is involved which in some case is not even open source. This represent a significant barrier as detailed knowledge of the job at hand and great patience is needed for achieving the goal.

2) Memory Dump

All the information extraction happens on the volatile memory dump that is obtained from the running virtual machine instance. This represents a large overhead for two reasons.

a) One copying the whole data from the RAM and writing it to the disk causes a lot of latency due to the GBs of data involved.
b) Copying the data causes the memory to be used and hence it slows down the virtual machine instance.

For coping with the overhead, Grizzly's design is such that the dump is not taken for each schema, instead it is taken after a fix interval with a pre-defined frequency. This frequency is tunable and can be set according to the threat environment in which Grizzly is to be used. But, by taking the dump at different intervals we tradeoff accuracy for efficiency.

3) Language Constraints

Grizzly's grammar is not fully developed and there are many useful aspects that are present in modern languages e.g lambda functions that can be quite beneficial in defining schemas for Grizzly and will allow for more indepth dense along with allow new vectors of attacks to be detected.

Future Work

The future work for Grizzly entails the following :

1) Language extension

Grizzly currently lacks the recently introduced changes in modern programming languages like lambda functions which constrains the schemas that can be written for the platform. Hence in the future, the language will be expanded and different useful programming techniques will be incorporated into Grizzly's grammar.

2) Compatibility

Currently we tested our system on Windows 7 with Debian 14 as the VMi performing VM on Xen hypervisor. Although there was great care taken for preventing system specific implementation and design there may be issues with configuration and information extraction from the dump on other systems especially old and less used systems. In future, we will work on increasing the compatibility of Grizzly for increasing the its effective and deployment.

3) Semantic Gap

By incorporating more and more memory forensics tools and other useful techniques, the semantic gap will be reduced which will lead to richer information available for analysis.

4) Automation

We also plan to make Grizzly a fully automated platform for VMI and for this we plan to make the platform in such a manner that each parameter can be tuned. Users will submit the schemas they want to run, their frequency and other related parameters. Then after binding the schema to Grizzly, it will automatically take care of all the complexity. Also the configuration parameters like output and dump frequency etc will be made tunable.

5) Overhead Minimization:

As mentioned in the limitation of Grizzly, the overhead of obtaining the dump from a running virtual machine instance is quite exorbitant due to which it has to be taken at different intervals which in turn reduces the system's effectiveness. In future we aim to eliminate this limitations by working directly with the RAM and hence not even taking the dump. By using copy on write mechanism and techniques from taking memory snapshots we aim to work directly with the volatile memory.

Conclusion

In our senior year project we have researched, designed and implemented Grizzly which is an automated malware detection framework based up virtual machine introspection. VMI gives us the ability to gather accurate information from the outside the system due to which it makes uses of the advantages of both HIDS and NIDS while avoiding their disadvantages. Grizzly provides automated signature based malware detection based upon VMI where the signature can be defined in a high level language and hence the complexity is reduced as it is delegated to the framework. We believe that such frameworks can be deployed as as service in public clouds for malware detection using VMI.

References

1. PAYNE, B., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In Security and Privacy, 2008. SP 2008. IEEE Symposium on (May 2008), pp. 233–247.

2. http://www.ieee-security.org/TC/SP2011/PAPERS/2011/paper019.pdf

3. http://libvmi.com/docs/gcode-intro.html

4. http://wiki.xen.org/wiki/Xen_Project_Software_Overview

5. B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009. [12] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys (CSUR), 44, 2012

6. https://www.sec.in.tum.de/assets/Uploads/scalability-fidelity-stealth.pdf

7. http://www.rekall-forensic.com/docs/Manual/

8. http://cyboxproject.github.io/documentation/

9. Rainer Wichmann. Samhain: distributed host monitoring system. http://samhain.sourceforge.net. M. J. Ranum. Intrusion detection and network forensics. USENIX Security 2000 Course Notes.

10. JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerabilityspecific predicates. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (New York, NY, USA, 2005), SOSP ’05, ACM, pp. 91–104.