Enhancement of Security Mechanism In Virtualization Environment

A Thesis submitted to Gujarat Technological University

For the Award of

Doctor of Philosophy

in

Computer/IT Engineering

by

Darshan Mansukhbhai Tank
Enrollment No.: 149997107002

Under Supervision of

Dr. Akshai Aggarwal

Under Co-supervision of

Dr. Nirbhay Kumar Chaubey

GUJARAT TECHNOLOGICAL UNIVERSITY
AHMEDABAD

[JANUARY – 2022]

Enhancement of Security Mechanism In Virtualization Environment

A Thesis submitted to Gujarat Technological University

For the Award of

Doctor of Philosophy

in

Computer/IT Engineering

by

Darshan Mansukhbhai Tank
Enrollment No.: 149997107002

Under Supervision of

Dr. Akshai Aggarwal

Under Co-supervision of

Dr. Nirbhay Kumar Chaubey

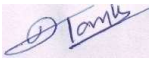
GUJARAT TECHNOLOGICAL UNIVERSITY
AHMEDABAD

[JANUARY – 2022]

DECLARATION

I declare that the thesis entitled **"Enhancement of Security Mechanism in Virtualization Environment"** submitted by me for the degree of Doctor of Philosophy is the record of research work carried out by me during the period from **May 2015** to **September 2021** under the supervision of **Dr. Akshai Aggarwal** and the co-supervision of **Dr. Nirbhay Kumar Chaubey** and this has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other institution of higher learning.

I further declare that the material obtained from other sources has been duly acknowledged in the thesis. I shall be solely responsible for any plagiarism or other irregularities if noticed in the thesis.

Signature of Research Scholar:  Date: 15/01/2022

Name of Research Scholar: **Darshan Mansukhbhai Tank**

Place: **Rajkot**

CERTIFICATE

I certify that the work incorporated in the thesis "**Enhancement of Security Mechanism in Virtualization Environment**" submitted by **Mr. Darshan Mansukhbhai Tank** was carried out by the candidate under my supervision/guidance. To the best of my knowledge: (i) the candidate has not submitted the same research work to any other institution for any degree/diploma, Associateship, Fellowship or other similar titles (ii) the thesis submitted is a record of original research work done by the Research Scholar during the period of study under my supervision, and (iii) the thesis represents independent research work on the part of the Research Scholar.

Signature of Supervisor: Date: 15/01/2022

Name of Supervisor: **Dr. Akshai Aggarwal**

Place: **Windsor, Canada**

Course-work Completion Certificate

This is to certify that Mr. **Darshan Mansukhbhai Tank** Enrollment no. **149997107002** is a PhD scholar enrolled for the PhD program in the branch **Computer/IT Engineering** of Gujarat Technological University, Ahmedabad.

(Please tick the relevant option(s))

☐ He has been exempted from the course-work (successfully completed during M.Phil. Course)

☐ He has been exempted from Research Methodology Course only (successfully completed during M.Phil. Course)

☑ He has successfully completed the PhD coursework for the partial requirement for the award of a PhD Degree. His performance in the course work is as follows-

Grade Obtained in Research Methodology (PH001)	Grade Obtained in Self Study Course (Core Subject) (PH002)
BC	AB

Signature of Supervisor: Date: 15/01/2022

Name of Supervisor: **Dr. Akshai Aggarwal**

Place: **Windsor, Canada**

Originality Report Certificate

It is certified that a PhD Thesis titled **"Enhancement of Security Mechanism in Virtualization Environment"** by **Darshan Mansukhbhai Tank** has been examined by us. We undertake the following:

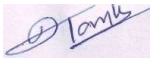
a. The thesis has significant new work/knowledge as compared to already published or are under consideration to be published elsewhere. No sentence, equation, diagram, table, paragraph, or section has been copied verbatim from previous work unless it is placed under quotation marks and duly referenced.

b. The work presented is original and the own work of the author (i.e., There is no plagiarism). No ideas, processes, results or words of others have been presented as Author own book.

c. There is no fabrication of data or results which have been complied/analysed.

d. There is no falsification by manipulating research materials, equipment or processes, or changing or omitting data or results such that the research is not accurately represented in the research record.

e. The thesis has been checked using "**URKUND Plagiarism Checker**" (copy of originality report attached) and found within limits as per GTU Plagiarism Policy and instructions issued from time to time (i.e., permitted similarity index <=10 %).

Signature of Research Scholar:  Date: 15/01/2022

Name of Research Scholar: **Darshan Mansukhbhai Tank**

Place: **Rajkot**

Signature of Supervisor:  Date: 15/01/2022

Name of Supervisor: **Dr. Akshai Aggarwal**

Place: **Windsor, Canada**

Copy Originality Report

Curiginal

Document Information

Analyzed document	Final_Thesis_Darshan_M_Tank.pdf (D124841138)
Submitted	2022-01-13T17:36:00.0000000
Submitted by	Darshan Tank
Submitter email	dmtank@gmail.com
Similarity	3%
Analysis address	dmtank.gtuni@analysis.urkund.com

Sources included in the report

W	URL: https://www.xn--neellco-cvb.com/stuff/NLP/Itzik%20Kotler%20-%20Process%20Injection%20Techniques%20Gotta%20Catch%20Them%20All%20-%20DEF%20CON%2027%20Conference.en.txt.vtt2text.txt Fetched: 2022-01-13T17:36:24.7070000	1
W	URL: https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/barabosch_quincy_dimva2017.pdf Fetched: 2021-03-28T16:29:06.1970000	7
W	URL: https://quizlet.com/233507926/memory-forensic-flash-cards/ Fetched: 2021-12-31T13:08:21.1730000	1
W	URL: https://icegrave0391.github.io/2020/03/10/memfor-8/ Fetched: 2021-09-24T19:50:15.3700000	4
W	URL: https://dokumen.pub/learning-malware-analysis-9781788392501-1788392507.html Fetched: 2021-12-30T00:20:25.5030000	2
W	URL: https://www.exploit-db.com/docs/english/13007-reflective-dll-injection.pdf Fetched: 2021-05-16T13:56:47.1600000	1
W	URL: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process Fetched: 2020-04-26T15:23:52.1730000	5
W	URL: https://github.com/CptGibbon/Windows-Process-Injection Fetched: 2021-05-16T13:57:02.9470000	4

Signature of Research Scholar

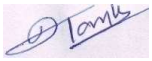
Signature of Supervisor

PhD Thesis Non-Exclusive License to GUJARAT TECHNOLOGICAL UNIVERSITY

In consideration of being PhD Research Scholar at GTU and in the interests of the facilitation of research at GTU and elsewhere I, **Darshan Mansukhbhai Tank** having Enrollment No. **149997107002** hereby grant a non-exclusive, royalty-free and perpetual license to GTU on the following terms:

a) GTU is permitted to archive, reproduce and distribute my thesis, in whole or in part, and/or my abstract, in whole or in part (referred to collectively as the "Work") anywhere in the world, for non-commercial purposes, in all forms of media;
b) GTU is permitted to authorize, sub-lease, sub-contract or procure any of the acts mentioned in paragraph (a);
c) GTU is authorized to submit the work at any National/International Library, under the authority of their "Thesis Non- Exclusive License";
d) The Universal Copyright Notice (©) shall appear on all copies made under the authority of this license;
e) I undertake to submit my thesis, through my university, to any Library and Archives. Any abstract submitted with the thesis will be considered to form part of the thesis.
f) I represent that my thesis is my original work, does not infringe any rights of others, including privacy rights, and that I have the right to make the grant conferred by this non-exclusive license.
g) If third part copyrighted material was included in my thesis for which, under the terms of the Copyright Act, written permission from the copyright owners is required, I have obtained such permission from the copyright owners to do the acts mentioned in paragraph (a) above for the full term of copyright protection.
h) I retain copyright ownership and moral rights in my thesis and may deal with the copyright in my thesis, in any way consistent with rights granted by me to my university in this non-exclusive license.
i) I further promise to inform any person to whom I mat hereafter assign or license my copyright in my thesis of the rights granted by me to my university in this non-exclusive license.

j) I am aware of and agree to accept the conditions and regulations of a PhD, including all policy matters related to authorship and plagiarism.

Signature of the Research Scholar: 

Name of Research Scholar: **Darshan Mansukhbhai Tank**

Date: 15/01/2022 Place: **Rajkot**

Signature of Supervisor: 

Name of Supervisor: **Dr. Akshai Aggarwal**

Date: 15/01/2022 Place: **Windsor, Canada**

Seal:

Thesis Approval Form

The viva-voce of the PhD Thesis submitted by **Mr. Darshan Mansukhbhai Tank** (Enrollment No. **149997107002**) entitled "**Enhancement of Security Mechanism in Virtualization Environment"** was conducted on **15-01-2022** via **Online Platform** from 02:00 PM (IST) onwards.

(Please tick any one of the following options)

☑ The performance of the candidate was satisfactory. We recommend that he be awarded a PhD degree.

☐ Any further modifications in research work recommended by the panel after 3 months from the date of the first viva-voce upon request of the Supervisor or request of Independent Research Scholar after which viva– voce can be re-conducted by the same panel again.

(Briefly specify the modifications suggested by the panel)

☐ The performance of the candidate was unsatisfactory. We recommend that he should not be awarded a PhD degree.

(The panel must give justifications for rejecting the research work)

Dr. Akshai Aggarwal
Supervisor (Name and Signature)

Dr. Deepak Garg
External Examiner - 1 (Name and Signature)

Dr. H. L. Mandoria
External Examiner - 2 (Name and Signature)

Dr. Kalpdrum Passi
External Examiner - 3 (Name and Signature)

Abstract

Virtualization is a vital innovation to empower distributed computing. Virtualization makes and runs different Virtual Machines (VMs) on a solitary physical machine utilizing Virtual Machine Monitor (VMM). Security of VMs is a significant concern with the virtualization environment. Virtual machines are an essential target for an adversary to get unscrupulous admittance to the organization's virtual infrastructure. As malware turned out to be progressively more persistent and sophisticated, traditional security countermeasures and innovations utilized for securing virtualized environments are not equipped for cutting-edge malware discovery. Regardless of late advances in malware detection ability, it is as yet workable for a likely adversary to remain unfound.

Today's progressed malware can easily stay away from recognition by adopting a few avoidance methodologies. Process injection is one such procedure to avoid recognition from security products since the execution is covered under a genuine process. Process injection has been an amazing arm in the enemy's arsenal for quite a while and one of the compelling tools in the aggressor's tool compartment. An adversary can use different process injection strategies to inject malicious code into a remote running process, sidestep process level limitations, and admittance to context explicit information that would somehow not be able to be achieved. There are different ways to inject code into a victim process and these implementations exist for each major operating system however are conventionally platform explicit.

Malevolent exercises are regularly implemented by injecting malicious code into running processes, which is frequently imperceptible by traditional anti-malware strategies. Diverse code injection procedures are used by malware to gain secrecy and to sidestep security products or tools. We covered eight distinct executions of process injection in this work. We completed dynamic malware examination utilizing an incredible memory investigation framework called Volatility Framework. We run Windows 7, Windows 8.1, and Windows 10 as guest OS on virtual machines as all three are the most widely utilized windows versions. We have executed a module utilizing Volatility (an open-source memory forensics framework) and adequately tried it on live VMs and malware-contaminated memory images.

Our primary focus in this research work is to propose an altogether new out-of-VM approach based on advanced memory introspection to identify process injection of varied types in a virtualized environment. We present an agentless solution where we screen malware running inside VM's memory from outside the VM. Trial results show that our model characterizes injected memory regions with high accuracy and completeness and has more true positives and fewer false positives when contrasted with other existing frameworks. Our proposed detection approach assures exact and reliable outcomes and precisely pinpoints injected memory sections. Rather than recognizing tainted processes, our proposed framework distinguishes a real malicious memory region in the virtual location space of a contaminated process. Our proposed framework recognizes more malware families and dominates the other methodologies in all assessment metrics. One can likewise perform live introspection of running VMs for a conceivable sign of code injection.

This work is intended to automate the detection of different process injection techniques in a virtualized environment and dump recognized malicious memory regions to disk for a detailed analysis and assessment. This PhD work would be useful for a software company that works in the area of security at the Infrastructure as a Service layer in the cloud computing model for possible integration of the solution in their Anti-Malware product.

Keywords: Malware Detection, Memory Analysis, Process Injection, Security, Virtual Machine Introspection, Volatility, Windows

Acknowledgement

First and foremost, I would say thanks to God Almighty Who has given me strength, positive energy, and persistence to complete this thesis. I would like to thank my parents, Jayaben Mansukhbhai Tank and Mansukhbhai Anandbhai Tank for supporting me, cherishing me and empowering me at each progression of my life.

I would like to acknowledge my obligation and render my warmest gratitude to my respectable Ph.D. supervisor, Dr. Akshai Aggarwal, Former Vice-Chancellor, Gujarat Technological University, India, and Professor Emeritus, University of Windsor, Canada, who made this work conceivable. His cordial direction and expert guidance have been invaluable throughout all phases of the work.

I would wish to offer my heartiest thanks to my Ph.D. Co-Supervisor, Dr. Nirbhay Kumar Chaubey, Dean of Computer Science, Ganpat University, Gujarat for broadened conversations and significant ideas which have contributed incredibly to the improvement of the thesis. I would like to express my sincere thanks to my International Co-Supervisor, Dr. Om Prakash Vyas, Professor, Department of IT, IIIT-Allahabad for his continuous support and kind guidance throughout the tenure of my research.

I would like to extend my sincere thanks to my Doctoral Progress Committee members Dr. Savita Gandhi, Professor & Head, Department of Computer Science, Gujarat University, Ahmedabad, and Dr. Vipul K Dabhi, Associate Professor, Department of Information Technology, Dharmsinh Desai University, Nadiad, for their significant remarks, valuable suggestions and support to visualize the problem according to the alternate point of view. Their humble approach and the way of appreciation for great work have consistently created an amenable environment and boost-up my confidence to push the limit. I also acknowledge Honourable Vice-Chancellor, Registrar, Controller of Examination, Dean of Engineering and staff members of Ph.D. Section of Gujarat Technological University for administrative assistance and kind help.

Special thanks to my better half, Poonam Tank for her continued love and support. I greatly value her understanding during the entire duration of my Ph.D. programme. Thank you for supporting me for everything. The expressions of appreciation go to my beloved daughter Prarthana and son Aniruddh for compromising my love during the period of thesis writing.

Last however not least; I would like to thank every one of my companions and associates who have straightforwardly and by implication helped me in the culmination of this study.

Darshan M. Tank

Table of Contents

List of Abbreviations

VMs	Virtual Machines
VMM	Virtual Machine Monitor
VMI	Virtual Machine Introspection
API	Application Programming Interface
KVM	Kernel-based Virtual Machine
LibVMI	VMI Library
APC	Asynchronous Procedure Call
OS	Operating System
DLL	Dynamic Link Library
IAT	Import Address Table
PE	Portable Executable
IaaS	Infrastructure as a Service
VAD	Virtual Address Descriptor
VMIPID	VMI-based Process Injection Detection
PoC	Proof of Concept
RAM	Random Access Memory
QEMU	Quick EMUlator

List of Figures

List of Tables

CHAPTER – 1

1. Introduction

1.1 Introduction

Distributed computing has turned into a prevailing computing paradigm in recent years. Virtualization is a key underlying technology to empower cloud computing. Virtualization creates and runs multiple Virtual Machines (VMs) on a single physical machine using Virtual Machine Monitor (VMM). The most alarming threat to the cloud is virtual machine security. Virtual machines are a primary target for an adversary to acquire unethical access to the organization's virtual infrastructure. As malware became increasingly more persistent and sophisticated, traditional security countermeasures and technologies employed for securing virtualized environments are not competent for advanced malware detection. Despite recent advances in malware detection capability, it is still possible for a potential adversary to stay unfound. Process injection is one such strategy to evade detection from security products since the execution is masked under a legitimate process. Process injection is a technique for executing subjective code in the address space of a different authentic process. An injecting process (injector) gains admittance to the target process's network resources, memory space, or system and likely elevated rights [1].

Process injection is an adaptable method that works with a wide scope of activities. Process injection techniques have a broad domain and account for many different kinds of behaviors. Attackers and malware frequently employ various process injection techniques. Process injection enables adversaries to get away from defensive controls by executing potentially malicious code in the context of apparently gentle ones. Executing code in the context of another kind process may also allow access to that process's memory, system, or network resources, and likely elevated rights. There are many different variants of process injection techniques - Remote DLL (Dynamic Link Library) Injection, Remote Thread Injection, PE (Portable Executable) Injection, Hollow Process Injection, Reflective DLL Injection, APC (Asynchronous Procedure Call) Injection, Thread Execution Hijacking, and AtomBombing etc. As there exists no direct access mechanism to the physical memory of VMs in a virtualized environment, detection of process injection is very challenging on VM as compared to a physical machine.

The motivation for this concept is to propose procedures to recognize malware in the form of code injection exploitation in virtual machine memory. This work focuses on eight different process injection techniques: Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, Portable Executable Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. There have been malicious programs executed on Windows 7, Windows 8.1 and Windows 10 virtual machines. The Volatility framework and LibVMI python bindings are used for dynamic malware analysis.

1.2 Virtualization Security

The most important objective of virtualization technology in cloud computing is to detach cloud-based users' environments. As a promising technology, it plays a very significant role in a cloud environment by supplying the competence of running numerous operating systems and applications on top of the same underlying computer hardware platforms [2]. Virtualization in cloud computing can be categorized into three different types based on their characteristics i.e., server, storage, and network virtualization. The cloud service provider must ensure the security of its infrastructure by addressing the security issues confronted by the elements of a virtualization platform.

The standard security agencies in computing have developed several policies, guidelines, recommendations, and best practices to protect the computing environment against potential security threats. Recently National Institute of Standards and Technology (NIST) released security recommendations for hypervisor deployment on servers, a report that provides recommendations on ensuring the secure execution of baseline functions of hypervisors, enabling multiple computing stacks called virtual machines to be run on a single physical host [3]. Cloud Security Alliance (CSA) released a whitepaper in 2015 on virtualization security best practices which provide direction on the recognition and administration of security threats particular to compute virtualization technologies that run on server hardware.

European Network and Information Security Agency (ENISA) released a report in 2017 on the security aspects of virtualization. This report provides an analysis of the status of virtualization security. ENISA presents current efforts, emerging best practices and known security gaps, discussing the impact the latter have on environments based on virtualization technologies. Information Systems Audit and Control Association (ISACA) provides the

virtualization security checklist intended for use with enterprise full virtualization environments. This checklist is also intended to be product and vendor agnostic to provide the broadest coverage possible about full virtualization security issues.

1.2.1 Malware Analysis

The European Network and Information Security Agency (ENISA), in its yearly threat landscape report, says that the most common cyber threats are malicious programs. Some Anti-Virus (AV) sellers distinguished more than four million examples of threats each day [4][5]. This figure shows how important malware analysis is to anyone who responds to computer security incidents.

Academicians, researchers and security experts have discovered many ideas and prototypes for malware detection and classification. Malware examination is the ability to cut out malware to understand how it works, how it affects the normal execution of target process, what API/System calls it calls, how it alters various data structures of operating system, and how it influences the behavior of a binary. The malware analysis approaches can be categorized into two classes: Signature-based static analysis and Behavioral-based dynamic analysis. Static analysis is accomplished without executing the samples while dynamic analysis is carried out by executing samples in the virtualization environment. The motivation behind the static investigation is to distinguish the malware before its execution while dynamic examination endeavors to identify pernicious conduct during or after the malware execution.

1.2.2 Memory Forensics

Memory is the best place to identify malicious software activities. Analysis of data embedded in memory is also called memory forensics. Memory forensics involves analysis of the physical memory contents of a computer system to derive information such as the source of a malfunction or malicious activity. Recent actions can be analyzed and tracked. Evidence not found elsewhere can be collected, e.g., memory malware only [6]. Memory analysis can be done in a live program, but can also be done in the memory dump. Disposal of memory is a brief snapshot of computer memory data from a particular time. It is best to find code injection using memory forensics.

In cases where malware encrypts the hard disk and does not permit the user to access the system at all, as seen in cases of ransomware [7], the only option left is memory forensics. In the case of the WannaCry malware, researchers have been able to extract the decryption key from the system's memory [8][9].

1.3 Research Objectives

This work aims to detect process injection and malware of varied types in a virtualized environment by employing an advanced memory introspection technique. The Ph.D. research work proposes to achieve the following objectives:

- To use Virtual Machine Introspection (VMI) for extracting data from the primary memory (RAM) of a virtual machine via the KVM hypervisor.
- To detect malware running in the form of process injection aggression inside the memory of virtual machines.
- To monitor the runtime state of VMs completely outside the VMs, from the hypervisor.
- To perform live introspection of running VMs for possible indication of process injection.
- To perform dynamic malware analysis by noticing the act of the injector while it is operating on a virtualization platform.
- To design, develop, and implement an algorithm that looks for injected memory regions in all active processes' memory.
- To categorize process's memory segments as either injected or benign.
- Accurately locate all injected memory regions across multiple targeted processes.

1.4 Scope of the Work

- To propose a novel approach incorporating capabilities of all major types of process injection detection.
- To utilize the Infrastructure as a Service (IaaS) cloud model as an anticipated attack environment.
- To develop, implement, and validate the proposed framework with defined evaluation metrics, i.e., precision, recall, false-positive rate, accuracy, and F1-score.

1.5 Research Motivation

Today's advanced malware makes use of different process injection techniques to either manipulate other running benign processes or to hide its existence. There exists no single system/solution which can detect all major types of process injection. Current code injection recognition frameworks or tools can't adapt to the current injection methods and be unsuccessful to uncover existing malware using certain concealing procedures and pinpointing accurate injected/malicious memory segments [10].

This work proposed a novel dynamic malware analysis framework using an advanced memory introspection approach to detect process injection of varied types in a virtualized environment. Although there are numerous process injection techniques, this work focuses on the detection of eight different implementations of process injection: namely, Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, Portable Executable Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. Instead of detecting infected processes, our goal is to detect an actual malicious memory region within the infected processes thereby pinpointing the malware exactly.

1.6 Problem Definition

Distributed computing has turned into a prevailing registering worldview in recent years. Virtualization is a key underlying technology to empower cloud computing. The most alarming security concern to the cloud is to protect an organization's virtual infrastructure. Virtual machines are a primary target for an adversary to acquire unethical access to the organization's virtual infrastructure. Traditional security countermeasures are inadequate to ensure virtual machine security in cloud computing. Today's advanced malware utilizes different process injection techniques to either manipulate other running processes or to hide its existence. There exists no single system/solution which can detect all major types of process injection.

This work proposes a novel Virtual Machine Introspection (VMI) based approach integrated with an advanced memory analysis framework to acquire higher-level semantic information from live memory data in VMs. We put forward a dynamic malware analysis framework using advanced memory introspection to detect process injection of varied types in a virtualized environment. Although there are numerous process injection techniques, this

work focuses on the detection of eight different implementations of process injection: namely, Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, Portable Executable Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. Instead of detecting infected processes, we aim to detect an actual malicious memory region within the infected process thereby pinpointing the malware exactly. The problem definition is:

"To detect process injection of varied types in a virtualized environment by employing advanced memory introspection approach."

1.7 Research Contributions

This work proposes an advanced memory introspection approach to detect process injection and malware of varied types in a virtualized environment. The significant contributions of the research are summarized as follows:

- An algorithm that scans the memory of VMs to locate malicious/injected memory regions in real-time.
- A Volatility plugin (ProcInjectionsFind) that implements the algorithm and prints information about the injected/victim process's VADs, hex dump and disassembly information at VADs base address.
- A Volatility plugin (ProcInjectionsFind) can be independently called and run against malware infected memory images or memory of live VMs and examine each memory region of all running processes to conclude if it is the result of process injection.
- The Volatility plugin (ProcInjectionsFind) can also dump the entire malicious/injected memory region (VAD) to disk for more detailed analysis.
- Experimental result analysis of the proposed framework shows the classification of injected memory regions with high accuracy and completeness and has more true positives and fewer false positives.
- We run different implementations of process injection targeting multiple processes. The proposed detection model recognizes injected memory regions into all infected processes.
- One can perform live introspection of running VMs for possible indication of process injection.
- This work covers eight different malware families of process injection.

1.8 Security Mechanisms and Challenges

To keep the virtualization environment secure, a security mechanism must be designed to detect, prevent or recover from security threats. The key to understanding where to place security mechanisms is to understand where physically in the environmental resources are deployed and consumed, what those resources are, who manages the resources, and what mechanisms are used to control them.

A semantic gap is one of the most important problems in the virtualized environment. It refers to the lack of higher-level knowledge of guest OS's internals within the hypervisor. The semantic gap is a result of the independent design of Guest OS and hypervisor. Solving this problem not only helps to develop security and virtual machine monitoring applications but also benefits hypervisor resource management and hypervisor-based service implementation [11].

Threads and Virtual Address Descriptors (VADs) are important data structures in process memory and sources of useful information regarding the execution of code on a computer system. Process threads and VADs information can be used to detect malicious behavior or anomalies in process execution. In this work, we present a method using thread and VAD data structures in process memory to locate injected code. We implement a plugin in python using the open-source volatility tool and have tested it on live VMs as well as malware infected memory images.

1.9 The Organization of the Thesis

The main contents of the thesis are as follows.

Chapter 1 deals with the general introduction of the research work. It elaborates on the basics of virtualization security, i.e., malware analysis and memory forensics. The chapter highlights research objectives, research motivation and scope of the work. It also discusses significant contributions in this research work. We study various security mechanisms and challenges in this chapter.

Chapter 2 present a review of the state of the work on virtualization security. The review identifies various issues and solutions in virtualization security. It describes various approaches proposed by different researchers for the detection of process injection attacks. It also discusses a VMI-based approach for malware analysis.

Chapter 3 presents the necessary information about the VMI tool, volatility framework, and Windows OS internals. It explains the process injection techniques covered in this work. It describes the proposed detection model, methodology of research, experimental setup, and malware hiding technique covered in this research work.

Chapter 4 presents the evaluation of the proposed detection model and defines evaluations metrics. It summarizes derived results and discusses their implications. It also presents a comparison of evaluation metrics with existing approaches and finally ends with concluding remarks.

Chapter 5 presents a summary of the entire work carried out in this research and conclusions derived from the presented work and its results. It also explains the possible work to be carried out in the future.

CHAPTER – 2

2. Literature Review

2.1 Virtualization Security

As an integral part of most businesses, virtualization is becoming more prevalent in various sectors of society. The virtue of virtualization rests on its ability to cut down operational costs and to provide an effective means of managing Information Technology (IT) resources [12]. Virtualization has changed the landscape of technology and revolutionized computing capability. Virtualization has been widely adopted. An enterprise runs most of its workloads in a virtualization environment. A virtualized system has many advantages compared to traditional computing systems. The key benefit of virtualization is to reduce the overall operational cost.

Virtualization is a technique to separate multiple users on a single machine. Virtual Machines (VMs) share the underlying hardware and rely on the software level isolation provided by the hypervisor. The Hypervisor provides virtualization of hardware resources and thus enables multiple computing stacks called VMs to be run on a single physical host [3]. The sharing of hardware resources between multiple guest systems optimizes resource usage. However, it has been discovered and proved that this isolation is not impenetrable [13].

2.1.1 Taxonomy of Virtualization Security Issues

Several security threats, risks, and vulnerabilities exist in the present virtualization infrastructure that an adversary can utilize to infiltrate the security and privacy of the systems in cloud computing environments. One can classify security issues into three categories. Figure 2.1 show these defined categories and their vulnerabilities and risks.

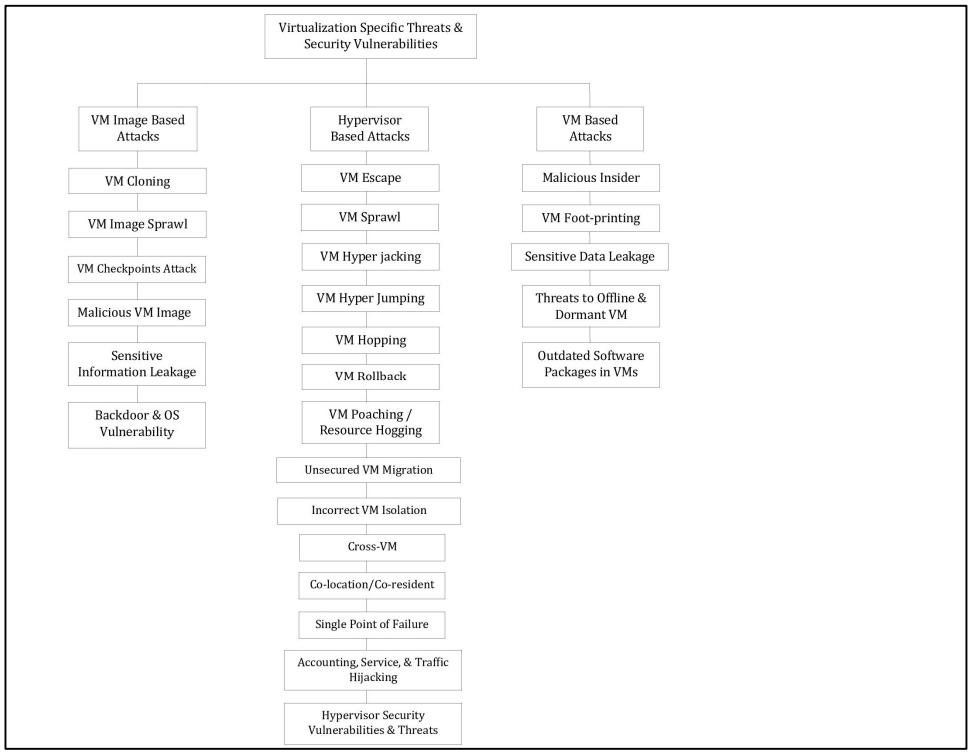


FIGURE 2.1 Taxonomy of virtualization specific security threats and vulnerabilities [14].

The authors have done a rigorous analysis of several peer-reviewed papers published in widely known international conferences and journals on security aspects of virtualization. The three dominant classes recognized were VM Image-based, Hypervisor-based, and VM-based attacks. The authors have also recognized possible sub-attacks at these categories and existing mitigation solutions to provide security to the virtualized environment. Table 2.1 shows three categories of cloud-based attacks on virtualization environments.

TABLE 2.1 Virtualization specific security threat, vulnerability, and mitigation technique [14].

Sr No	Attacks	Sub-Attacks	Existing Defense Mechanisms
1	VM Image-Based Attacks	VM Cloning VM Image Sprawl VM Checkpoint Attacks Malicious VM Image Sensitive Information Leakage Backdoor & OS Vulnerability	➢ Enforcing security policies and rules ➢ Encrypting the checkpoints ➢ Managing VM images ➢ Cryptographic techniques ➢ User awareness
2	Hypervisor-Based Attacks	VM Escape VM Sprawl VM Hyperjacking and VMM Rootkits VM Hyper Jumping VM Hoping VM Rollback VM Poaching/Resource Hogging Unsecured VM Migration Incorrect VM Isolation VM Information Leakage • Cross-VM • Co-location/Co-resident Single Point of Failure Account or Service Hijacking Unauthorized Access to Hypervisor (Insecure Hypervisor) Vulnerable cloud service provider's APIs (Insecure APIs) The workload of Different Trust Levels Hypervisor Security Vulnerabilities • SubVirt, Blue Pill, Vitriol Hypervisor Security Threats • Hypervisor Introspection • Hypervisor Alteration • Hypervisor Denial-of-Service	➢ Intrusion Detection/Prevention System (IDS/IPS) ➢ Hypersafe ➢ VMM patching ➢ Encrypting VMsR ➢ Proper configuration ➢ Intrusion Detection System ➢ Reducing the hypervisor attack surface ➢ Protecting hypervisor integrity ➢ Designing secure hypervisors ➢ Security-aware development of VMMs ➢ Hypervisor integrity checking and attestation ➢ Identifying and enforcing security policies
3	VM-Based Attacks	Inside VM attacks • Malware • Malicious insiders VM Foot-printing Sensitive data leakage within a VM Threats to offline and dormant VM Outdated software packages in VMs	➢ Using encryption and hashing of VMs state ➢ A Security-Conscious Scheduler for VMs ➢ Co-Residency Recognition via Side-Channel Analysis ➢ Constructing a MAC-based security framework ➢ Virtual Machine Monitor-Based Lightweight Intrusion ➢ Introspecting VMs

By doing the above analysis, one can conclude that attacks exploited on hypervisors have a much broader attack surface as compared to the VM-based and VM Image-based attacks.

2.1.2 Virtualization Security Solutions

Being a core component of cloud computing technology, cloud service providers must ensure the safety of their virtualized environment for the overall security of the cloud computing environment. Cloud service users may discuss and clear the security policies and standards with a cloud service provider before subscribing to their services. The virtualization element has a much larger attack surface than any other element of cloud computing. Cloud service models (i.e., IaaS, PaaS, and SaaS) offer different levels of security services. Various techniques have been proposed by different researchers to solve virtualization specific threats and vulnerabilities in the cloud computing environment. In this section, we review mitigation techniques and solutions proposed in the research articles for enhancing the security of virtualization components. In Table 2.2, we compile our observations of defense mechanisms.

TABLE 2.2 Comparison of various defense mechanisms.

Sr No	Author	Defense Mechanisms	Security Criteria						
			Securing Hypervisor	Securing VM	Securing VM Image	Data Confidentiality	Data Integrity	Data Availability	Access Control
1	Azab et al. [15]	HyperSentry	Y	-	-	-	-	-	Y
2	Wang & Jiang [16]	HyperSafe	Y	-	-	-	Y	-	-
3	Zhang et al. [17]	CloudVisor	Y	Y	-	-	Y	-	-
4	Keller et al. [18]	NoHype	Y	-	-	-	-	-	Y
5	Szefer & Lee [19]	HyperWall	-	Y	-	Y	Y	-	-
6	Xiong et al. [20]	CloudSafe	Y	-	-	Y	-	-	Y
7	Lee & Yu [21]	Virtualization Introspection System (VIS)	-	Y	-	Y	-	-	-
8	Kumara & Jaidhar [22]	Open-Source SECurity Event Correlator (OSSEC)	Y	-	-	-	-	Y	-
9	Götzfried et al. [23]	HyperCrypt	Y	-	-	Y	-	-	-
10	Tien et al. [24]	VM introspection	-	Y	-	-	-	-	-
11	Tang & Mi [25]	In-Hypervisor Memory Introspection (IHMI)	-	Y	-	Y	Y	Y	-
12	Zhang et al. [26]	VEDefender	Y	Y	-	-	-	-	-
13	Yadav & Challa [27]	A two-level security framework	-	Y	-	-	Y	-	-
14	Li et al. [28]	HypSec	Y	Y	-	Y	Y	-	-

2.2 Process Injection Detection

Process injection can be used with a wide range of activities. Many different kinds of behaviors are accounted for by process injection techniques. Various process injection techniques are used by attackers. There are three steps to true process injection, namely memory allocation, memory writing and code execution. Code injection allows adversaries to move away from protective controls by executing potentially harmful code. Executing code with regards to another benign process will allow admittance to that process's memory, network or system possessions.

There are many different process injection techniques, namely, Remote DLL Injection, Remote Thread Injection, PE Injection, Hollow Process Injection, Reflective DLL Injection, APC Injection, Thread Execution Hijacking, and AtomBombing etc. Detection of process injection in a virtual machine is very difficult as there is no direct access to the physical memory of the machine.

There exist many different approaches for the detection of process injection, such as monitoring process, Windows API (Application Programming Interface) calls, DLL/PE file events, system calls, and named pipes, etc. Typical injection methods can be found utilizing API call grouping patterns, for example, OpenProcess→ VirtualAllocEx→WriteProcessMemory→CreateRemoteThread [29]. Li et al. [30] presented a malware analysis model to analyze the process of a virtual machine. The model is based on a virtual machine introspection technology, which can monitor the program running in the virtual machine. Ajay Kumara et al. [31] presented Virtual Machine Introspection based malicious process detection approach for virtual machines. It is based on extracted system call information from introspected virtual machine memory pages during run state. To differentiate benign process from malicious one, system call information is employed.

Various methods exist for discovering possibly injected memory regions inside a process's virtual address space. One such technique is the notable module called Malfind for the Volatility memory analysis system. Volatility's Malfind plugin can detect injected memory sections [32]. Malfind does not detect DLLs injected into a process using CreateRemoteThread → LoadLibrary. Different types of process hollowing techniques get discovered with the Hollowfind volatility plugin [33]. However, these approaches have significant shortcomings. Malfind be affected by a high false positive rate, and Hollowfind

discloses only a subgroup of all admissible types of host-based code injection attacks. Membrane [34] reveals affected processes in place of actually malicious memory regions within a process. Although this decreases the size of the haystack to search in, it does not locate the malware precisely. Another method is the PowerShell Script Get-InjectedThread.ps1 by Jared Atkinson. Get-InjectedThread [35] filters each running thread to come to an end in case it is the aftereffect of memory injection.

There is a situation where the harmless utilization of Windows API calls might be normal and hard to recognize from unfriendly conduct. Most legitimate processes won't have to use Windows APIs such as 'CreateRemoteThread', so it is described as an extremely dubious API and is identified by numerous security items which may recognize the doubtful DLL on the disk [36]. The analytics that produced the majority of false positives came from looking for CreateRemoteThread calls from any process. Windows API calls that can be utilized to change memory inside another process, like VirtualAllocEx and WriteProcessMemory, might be used for the conceivable sign of code injection [36].

Barabosch et al. [37] presented a system called Quincy based on supervised machine learning to detect host-based code injection attacks in memory dumps. They have implemented Quincy for three Windows versions and released it as a Volatility plugin. Zhang et al. in [26] proposed a novel dynamic malware analysis framework called VEDefender that periodically scrutinizes the state of the introspected system to detect hidden, dead, and dubious processes in the monitored VM without modifying the guest OS kernel at the host level. Block et al. [38] introduced a novel approach to reveal all executable pages that are of potential interest for an investigator for a given user space process. Mathew et al. [39] proposed API call-based malware detection approach using a recurrent neural network - Long Short Term Memory (LSTM).

2.3 Virtual Machine Introspection (VMI)

Basic issues and state supported assaults have made criticalness for an alternate way to deal with programming security organizations. VMI is one such methodology utilized by security concerns. In its least complex terms, VMI is characterized as a strategy for investigating the runtime condition of a VM. Introspection can be achieved either from the hypervisor or from some virtual machine other than the one being supervised. VMI is an art of safeguarding a security-critical application running on virtual machines from security attacks [40]. VMI-

based approaches are widely adopted for security applications, software debugging, and systems management. One can introspect the VM from inside or outside of the VM. VMI-based tools may be located inside or outside of the VM. VMI tools can also be used for malware analysis to analyze the behavior of the malware and to detect the latest malware attacks. VMI coupled with existing virtual infrastructure management solutions can become a powerful tool for memory analysis and event correlation.

As the modern kernels are becoming very complex nowadays, hardening is a complex process and bugs can also lead to total system compromise, VMI offers a promising answer for various security-related issues and dangers. VMI offers better detachment, eliminates the dependence on the guest OS to work, and limits obstruction with the guest OS for the security applications. With VMI, one can likewise discover that Is there any possibly unsafe code being executed, without the utilization of in guest tools. Typical use cases of VMI are to investigate VM memory for infringement, i.e., code infusion, to ensure the integrity of in-guest security, and to monitor both kernel and user-mode processes.

2.4 Semantic Gap Problem in Virtualized Environment

The semantic gap is quite possibly the main problem in the virtualized environment and one of the primary restrictions of virtual machine introspection [41]. The semantic gap is a result of the independent design of guest OS and VMM. In a virtualized environment, the semantic gap can be defined as the extraction of high-level information of guest OS state from low-level information obtained externally at the hypervisor level [42]. We can do introspection within the virtual machines or outside the virtual machines. Security applications with VMI require observing the low-level interaction events between the guest OS and VMM to acquire useful information. By extracting low-level information from guest OS, VMM can recognize processes behavior running inside guest OS [42].

2.5 Summary of Literature Review

In summary, there exist several systems or solutions to detect process injection attacks in memory dumps, but they all suffer from the following major limitations.

- Bypass common detection methods.
- Detects memory regions with execute permission [89].
- Detects memory regions marked as private [89].
- Detects only a subgroup of process injections [90, 91, 92].

- Detects only a single injected code region.
- Unsatisfying detection rates, i.e., high false positive rate [89].
- Too coarse detection granularity, i.e., detects affected processes in place of actually injected memory regions within a process.
- Some of the existing plugins work correctly on memory image (dump) of specific Windows versions and may not work properly on 64-bit Windows guests, i.e., depends on hardware architecture.
- There exists no single system/solution which can detect all major types of process injection.
- Current process injection detection tools are not able to cope with the different implementations of process injection techniques and fail to accurately uncover existing malware employing certain hiding techniques [38].

Process injection detection systems could produce a notable amount of data. The produced data might not be directly beneficial for defense unless gathered under particular circumstances [43]. An evaluation data set should also match certain requirements. An evaluation data set should contain a considerable number of different families to evaluate the systems against different process injection techniques. We must ensure that our evaluation results are valid for recent malware families that run on the targeted Windows VMs. The data set should also contain goodware applications to estimate false positives. Malware classification systems should consist of a considerable number of samples [44]. As with every detection system, an adversary can try to understand and circumvent its detection heuristic. We still need more context to create an operational detection system. In this work, we overcome the limitations of the aforementioned systems by presenting the VMI-based Process Injection Detection (VMIPID) model. Our proposed process injection detection model can also uncover injected memory regions despite the mentioned malware hiding techniques.

CHAPTER – 3

3. Report on the Present Research

3.1 Background

This section provides a brief explanation of the virtualization environment, memory mapping under the hypervisor, memory forensics, and the significance of memory forensics.

3.1.1 Virtualization

Virtualization can be utilized to establish a segregated environment. It implies the formation of a virtual resource such as storage, file, operating system, desktop, server, and network. In OS-level virtualization, one can run numerous OSs on a solitary piece of hardware equipment. Virtualization innovation includes isolating the physical hardware and computer program by imitating hardware using a computer program. When a different operating system is working on top of the essential one, it is referred to as a Virtual Machine [45]. The computer that runs a hypervisor to emulate another computer on top of itself is known as a host and the emulated computer (VM) is a guest. In this thesis, we will use the terms guest and VM interchangeably.

Virtualization implies that a VM is created on a host computer that has access to the hardware [46]. Hardware is then emulated to make VMs believe they are running directly on the hardware. The software, firmware, or hardware that creates and manages the VMs is called the hypervisor or Virtual Machine Manager (VMM). Hypervisors can be classified into two categories [46], which are illustrated in Fig. 3.1;

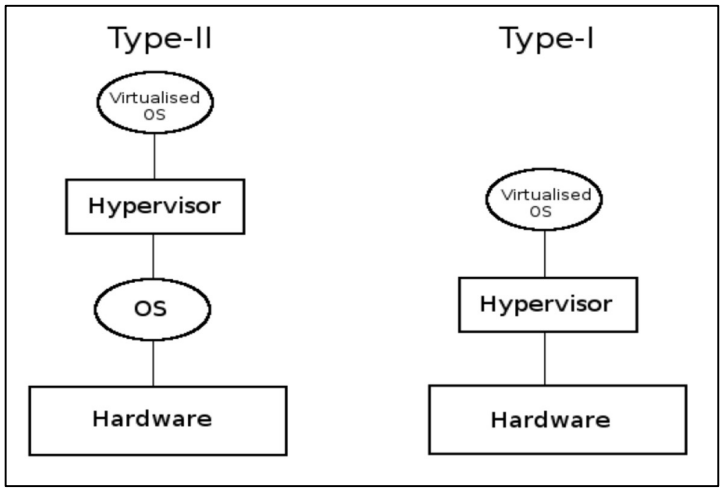


FIGURE 3.1 Hypervisor Type-I and Type-II [47].

- Type - I: These hypervisors run directly on the hardware of the host and are disconnected from the host OS. Hypervisors that are Type-I includes Xen, Oracle VM Server, and Microsoft Hyper-V.
- Type - II: The hypervisor runs on the host OS like an application. The guest OS then runs on the host OS as a process. The two OSs are however abstracted from each other, i.e., they cannot see or affect what the other OS is doing.

Linux's Kernel-based Virtual Machine (KVM) is a hybrid hypervisor of both Type-I and Type-II. This is because KVM is a kernel module that converts the kernel to a Type-I hypervisor. KVM needs an emulator to emulate the hardware since it does not perform this by itself. This emulator is often Quick EMUlator (QEMU), and together QEMU and KVM can run VMs at near-native speeds. KVM is also capable of so-called Nested Virtualization, i.e., that a VM is hosting a VM [47].

3.1.2 Memory Mapping

In an ordinary situation, there are two degrees of memory: virtual memory and physical memory of the physical machine. In any case, when we talk about hypervisors, there are three degrees of memory: virtual memory and physical memory of the virtual machine, and physical memory of the host machine. The hypervisors just allot memory to the virtual machine. Of course, hypervisors have no information on what sorts of exercises are being performed inside the virtual memory of the virtual machine. To get that data, extra tools must be installed. The following is a generalized illustration of memory-sharing inside the virtual machine. Fig. 3.2 shows the three degrees of memory tending to under hypervisor [48].

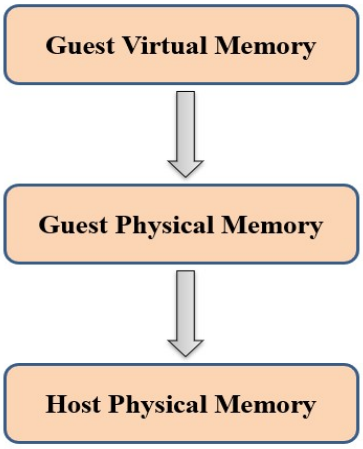


FIGURE 3.2 Memory mapping under the hypervisor.

One of the essential goals of the VMI instruments is to decipher the memory locations of the virtual machine's virtual memory: first, from the virtual to the physical memory of the virtual machine, then, at that point, to the physical memory of the host machine. This will help the hypervisor to get to the right memory region during contemplation.

3.1.3 Why Memory Forensics?

The memory is the best place to identify malicious software activities. Everything in the operating system traverse through RAM (Random Access Memory). Data about each running process and thread, open records, erased documents, Windows registry keys and event logs are contained in the RAM.

Notice that while a malignant program is being executed, it can't be removed from memory, in contrast to the hard disk. Every function performed by an operating system or application brings about changes to the PC's memory which can stay for a long time, keeping up with them [49].

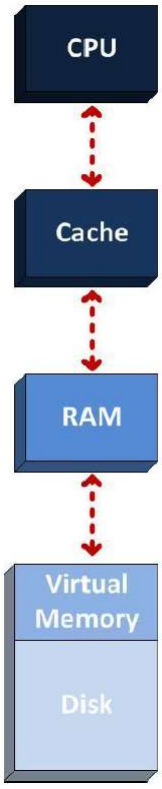


FIGURE 3.3 Operating system memory management.

The examination of information caught from memory is called memory forensics. A malware analyst can study the system configuration and identify inconsistencies in memory. Recent activities can be tracked and analyzed. Memory analysis can be done on the live system, and it should likewise be possible on a dump of the volatile memory. A memory dump is a preview catch of PC memory information from a particular moment. There are business memory examination apparatuses, as WindowsSCOPE Cyber Forensics [50], F-Response [51], Windows Memory Forensic Toolkit (WMFT) and open-source instruments like the Volatility Framework.

3.2 LibVMI - VM Introspection Tool

LibVMI is an open-source introspection library. LibVMI focuses on writing and reading memory from VMs. LibVMI is an extended version based on XenAccess Library. LibVMI is designed to work across multiple platforms [52]. LibVMI allows accessing the memory of running virtual machines in real-time. In addition to memory access, LibVMI also supports memory events. LibVMI can be used to connect the semantic hole between the hypervisor and guest operating systems [55]. It offers the following features.

- Easily extensible and optimized performance.
- It provides near-native speeds.
- Address the semantic gap problem.
- Access a VM's state from outside of the VM and broad platform support.

LibVMI is a virtual machine introspection library and is capable of performing VMI on KVM, Xen and QEMU hypervisors [55]. 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. LibVMI has the memory forensics framework called "Volatility" built-in via a python API. The python wrapper, according to the documentation, was able to join LibVMI with Volatility in a way that made live memory analysis possible.

3.3 The Volatility Framework

The Volatility Framework is an open assortment of tools, executed in Python under the GNU General Public License, for the extraction of digital artifacts from volatile memory (RAM) samples or memory images (dump). It is an open-source memory forensics framework for malware investigation and occurrence reaction [49]. It upholds investigation for Linux, Windows, Mac, and Android frameworks [53]. Different volatility commands are additionally open, created and kept up with by the community to extricate helpful data from memory samples. Volatility framework can be used as a memory forensic tool stash to distinguish progressed malware with a genuine case scenario [54]. The volatility structure offers the accompanying provisions.

- An advanced & open-source memory analysis tool.
- Support live analysis of virtual machines.
- Runs on Linux, Windows, Mac, and Android systems.
- It can be used to detect advanced malware with a real case scenario.
- Support a variety of file formats.
- Plugins can be developed and distributed independently.

The volatility tool supports a wealth of perceptions into the working of a system [55]. We used Volatility 2.6.1 in our research to extract higher-level semantic information from the live Windows virtual machines. The LibVMI also adds improved integration with Volatility [56]. We can connect Volatility with LibVMI to view the memory of running VMs from any

virtualization platform that LibVMI supports. If we would like to extract higher-level semantic information from running VMs, then we would recommend utilizing the LibVMI Python bindings and Volatility. The LibVMI Python bindings include a Volatility address space plugin that enables us to use Volatility on a live virtual machine. In this work, we leverage Virtual Machine Introspection (VMI) and memory forensic technology to analyze higher-level semantic information from live memory data in VMs.

3.4 Windows OS Internals

To follow the idea of this theory, essential information on the Windows operating framework internals is needed on the subjects as follows.

3.4.1 Processes

A process, in its simplest terms, is an executing program. An application consists of one or more processes. The _EPROCESS is the name of the data structure that Windows uses to represent a process.

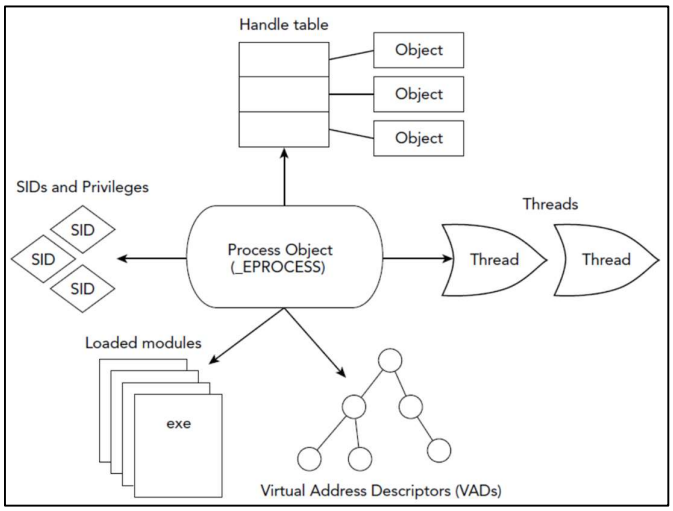


FIGURE 3.4 Basic process resources [49].

Fig. 3.4 shows the basic resources that belong to a process. Each process has its own private virtual memory space that is isolated from different processes. Inside this memory space, one can discover the process executable, its rundown of stacked modules, its stacks, heaps, and designated memory segments containing everything from user input to application-explicit information structures [49]. Windows sorts out the memory segments using virtual address descriptors (VADs).

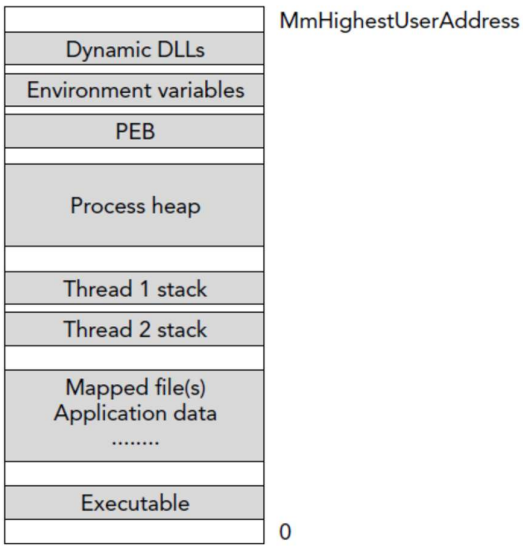


FIGURE 3.5 The typical contents of process memory [49].

A process contains its own independent virtual address space with both code and data, protected from other processes. Fig. 3.5 shows an address space layout of process memory. As depicted in Figure 3.5, an address space layout of process memory includes components like Dynamic linked libraries (DLLs), Environment variables, Process Environment Block (PEB), Process heaps, Thread stacks, Mapped files and application data, and Executable [49].

3.4.2 Threads

A thread (_ETHREAD structure) is the essential unit to which the OS designates processor time. A thread can execute any piece of the process code, including parts as of now being executed by another thread [57]. A thread is the fundamental unit of CPU usage and execution. A thread is described by a thread ID, CPU register set, and execution stack(s), which assist with characterizing a thread's execution setting. Despite their unique execution contexts, a process's threads share a similar code, information, address space, and operating system assets [49].

A process possesses at least a single thread of execution. A process with multiple threads can appear to be simultaneously performing multiple tasks. A process provides the execution environment, resources, and context for threads to run. In terms of memory forensics, thread data structures are useful because they often contain timestamps and starting addresses. This information can help you determine what code in a process has been executed and when it began [49].

3.4.3 Process Handles

Handles are normally referencing system objects; these objects can't be straightforwardly gotten to from user mode demands. Handles permit these objects to be referred to and certain system routines given to user mode will permit you to change or inquiry the information that they reference [58]. A handle is a reference to an open instance of a kernel object, similar to a file, registry key, mutex, process, or thread. There are near 40 unique sorts of kernel objects. By recognizing and contributing the specific objects a process was getting to at the hour of memory get, it is plausible to appear at different forensically applicable finishes [49]. Before a process can get to an object, it first opens a handle to the item. Exactly when a process is finished using an object, it should close the handle by calling the appropriate function.

3.4.4 Modules

Each active process comprises at least one module. A module is an executable file or DLL. A module can be specified by its base address in the target's virtual address space, or by its index in the list of modules the engine keeps up with for the target [59]. The first module is the executable file. In Windows operating system, the modules are addressed through the _LDR_DATA_TABLE_ENTRY structure.

Each active process has its list of stacked modules. There is just a single _LDR_DATA_TABLE_ENTRY structure for every module, each is linked in three unique orders. The LDR member of the process's PEB points to the process's _PEB_LDR_DATA which contains the list heads as InLoadOrderModuleList, InMemoryOrderModuleList and InInitializationOrderModuleList [60]. InLoadOrderModuleList addresses a linked list that arranges modules in the order in which they are stacked into a process. InMemoryOrderModuleList addresses a connected list that sorts out modules in the order in which they show up in the process' virtual memory design. InInitializationOrderModuleList addresses a linked list that sorts out modules in the order in which their DllMain function was executed [49].

3.4.4.1 Kernel32.dll

A kernel is the main part of an operating system that plays out essential tasks like memory management. Kernel32.dll is a Windows kernel module. It is a 32-bit dynamic link library that is utilized in Windows operating systems. On system boot-up, kernel32.dll is loaded

into a protected memory with the goal that it isn't undermined by other system or user processes. It runs as a background process [61]. LoadLibrary (or LoadLibraryEx) API function loads the predetermined module into the location space of the calling process. Processes call LoadLibrary or LoadLibraryEx to expressly link to a DLL. If the function succeeds, it maps the predetermined DLL into the location space of the calling process and returns a handle to the DLL [62].

3.4.5 Process' Import Address Table (IAT)

The Import Address Table (IAT) is a call table of user space modules. The executable modules running on Windows have at least one IAT incorporated as a part of their file structures [63]. The IAT is a part of the PE structure. The DLLs loaded utilizing LoadLibrary (or LoadLibraryEx) API call is unequivocally loaded at run-time [62] and there isn't any comparing entry in the process' IAT. Loaded DLLs put away in IAT are not viewed as dubious, in contrast with the other loaded DLLs.

3.4.6 Virtual Address Descriptors (VADs)

The VAD is one of the essential data structures utilized by processes that reside in kernel memory and contains data about adjoining process virtual location space distribution. A memory region (likewise called memory section or memory segment) is a bunch of successive pages inside a process. Virtual address descriptor is the term for a memory segment on Windows OS. VAD is utilized by the Windows OS memory administrator and is coordinated as a tree. The VAD tree design can be utilized as a process eye perspective on physical memory [64].

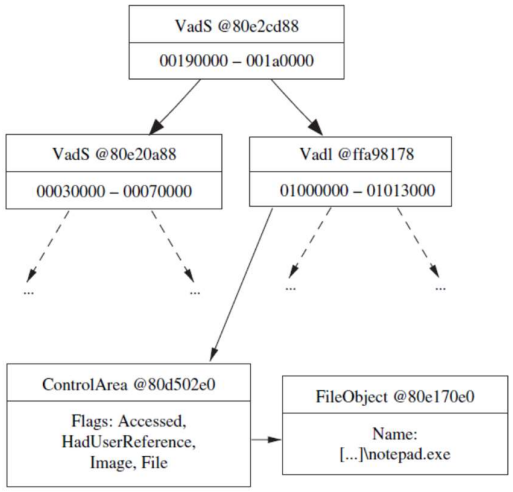


FIGURE 3.6 The VAD tree data structure [64].

As shown in Fig. 3.6, the VAD tree is a self-balancing binary tree that lists memory ranges assigned utilizing the VirtualAlloc() function call. At some random VAD nodes, memory tends to be lower than those contained at the current VAD node can be found in the left subtree and higher reaches can be found in the right subtree. Another entry is added to the VAD tree structure at whatever point a process utilizes the VirtualAlloc() or VirtualAllocEx() function call to allot another memory area [64]. The VADs contain data, for example, the starting and ending locations of the allotted memory block, memory-mapped document name, and the initial protection (read, write, execute) alongside a few different attributes.

The VAD fundamentally contains three kinds of nodes: in particular, VadS (_MMVAD_SHORT structure), Vad (_MMVAD structure) and VadL (_MMVAD_LONG structure), to store process-related data. The operating system chooses the node structure to make dependent on the API utilized for allocation. The parameters of the memory range characterized by the three nodes are put away in flags. A node can have at least one flags relying upon the utilization of the memory section. This data about process memory space makes VAD a significant data structure for a memory forensic examiner.

3.4.6.1 Walking through the VAD Tree

Every active process (_EPROCESS structure) points to the base of the VAD tree. The VadRoot addresses the root node of the VAD tree. It contains itemized data about a process' dispensed memory portions, including the original access consents (read, write, execute) and regardless of whether a file is mapped into the area. At the VadRoot node, follow each link to the left and right subtrees until the whole tree is crossed. The VadRoot.traverse() API can be utilized to produce a VAD object for every node in the tree.

3.4.7 Portable Executable (PE) File Format

The Portable Executable (PE) format is a document format for executables, object code, DLLs and others utilized in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that epitomizes the data essential for the Windows OS loader to deal with the wrapped executable code. On NT operating systems, the PE format is utilized for EXE, DLL, SYS (device driver), MUI and other file types [65]. The format of an operating system's executable file is in numerous ways a reflection of the operating system [66]. Microsoft has designed the Portable Executable (PE) file format for use by the

Windows family of operating systems. The name "Portable Executable" alludes to the way that the format isn't architecture explicit [67].

3.5 Process Injection Techniques Covered in this work

Malware can move away from its detection and sidestep the antivirus check in numerous ways. Process injection likewise called code injection used by the malware is one such approach to conceal its existence and to gain admittance to the victim process' memory space. There are various approaches to inject code into a target process. These implementations exist for each major operating system but are typically platform-specific. This section briefly describes different publicly known process injection methods, against which we assess our methodology in Section 4.3. We center around following eight distinctive process injection procedures. All strategies listed below are unnoticeable by a single security product or solution.

3.5.1 Remote DLL Injection Via CreateRemoteThread and LoadLibrary

Remote DLL injection is an injection technique that loads a malicious DLL inside the context of a running genuine (target) process. The remote process is exploited by CreateRemoteThread API and its memory content is changed. When the compromised process loads the malicious DLL, the OS (Operating System) consequently calls the DLL's DllMain function, which is characterized by the author of the DLL. This function contains the malicious code and has as much access to the system as the process in which it is running [68][69]. The essential is the presence of the malicious DLL in the disk. This method is considered the most straightforward injection strategy. A malware that utilizes this technique is Poison Ivy.

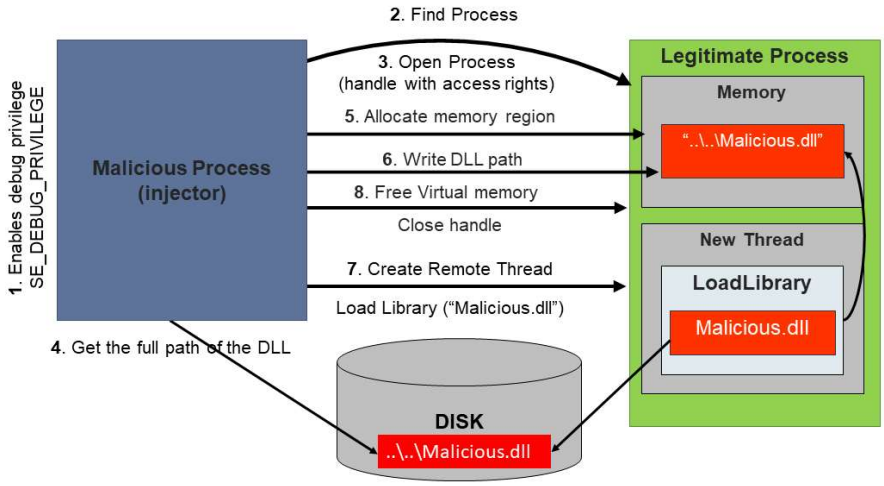


FIGURE 3.7 Remote DLL injection steps [70].

The steps that the malicious process acts in this method are as per the following:

- Enables debug privilege (SE_DEBUG_PRIVILEGE) that gives it the right to read and write in other process' memory as if it were a debugger.
- Finds the target process' ID utilizing its name.
- Opens the target process with the desired access rights and gets a handle to it.
- Gets the full path and name of the DLL, which as of now exists on the disk.
- Allocates a new memory region within the virtual address space of the target process of size as the length of the malicious DLL full name (including its path) and gets the memory address of this region.
- Writes the DLLs' full name into the newly allocated memory section utilizing the WriteProcessMemory API function at the address retrieved in the previous step.
- Creates a new thread inside the context of the genuine process that executes the LoadLibrary API function using as a parameter the written DLL full name.
- Cleans up by extricating the allocated memory and closing the handle on the target process and created thread.

3.5.2 Remote Thread Injection Using CreateRemoteThread

This is the easiest injection method. A malware injects malignant shellcode into another process and executes it as a thread. This technique works by injecting the shellcode also called payload into the context of another worthy process and makes a thread for that process

to run the payload. It opens the target process with all access and gets its handle, allocates memory in the target process, writes to it, and makes a remote thread employing CreateRemoteThread API.

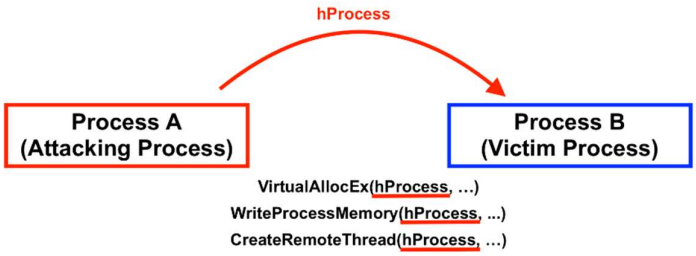


FIGURE 3.8 Remote thread injection [71].

3.5.3 PE Injection

A Portable Execution (PE) is a Windows file format for executable code. It is a data structure containing all the data required with the goal that Windows realizes how to execute it. Here the malware injects a malignant PE image into an already running process. This is a disk-less activity, i.e., the malware doesn't have to write its payload onto the disk before the injection.

FIGURE 3.9 PE injection [35].

3.5.4 Reflective DLL Injection

It is a library injection technique that performs the loading of a library (DLL) from memory into a target process. The course of distantly injecting a library into a target process includes two stages. In the primary stage, the library should be written into the address space of the target process and in the subsequent stage, the library should be loaded into the target process so that its run time assumptions are met. A pernicious process writes a DLL (as a sequence of bytes) into the memory space of a target process. The DLL handles its initialization without the assistance of the Windows loader. The DLL doesn't have to exist on the disk before being injected.

3.5.5 Hollow Process Injection

Hollow Process Injection (or Process Hollowing, or Process Replacement, or Dynamic forking) is a code injection strategy in which the hollowing process starts a new instance of a genuine process in a suspended state. The executable segment of the genuine process in the memory is swapped out (hollowed) and is supplanted with malignant code, generally malicious executable [33]. From that point forward, the authentic process is continued and it executes the malicious code for the rest of its process lifetime inside its legitimate context of the process. The hollowing process ordinarily follows the following steps [33]:

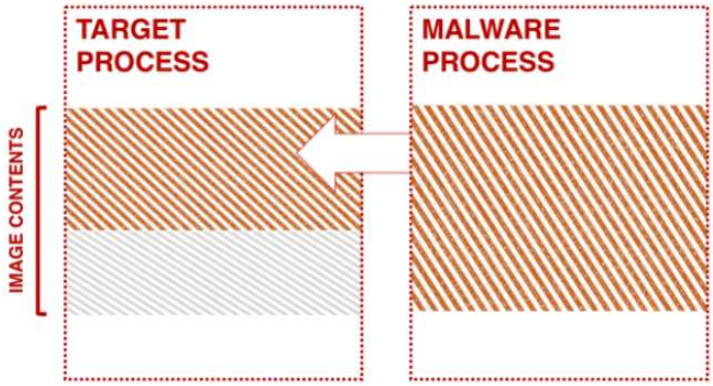


FIGURE 3.10 Process hollowing [35].

- Creates a new instance of the genuine process, but with its first thread suspended.
- Procures the malignant executable code to inject.
- Reads the legitimates' processes entry point and image base address, and afterwards free or unmap the containing memory segment.
- Allocates a new memory region inside the virtual address space of the authentic process.
- Copies the malicious PE header just as every PE section into the hollow made inside the legitimates' process memory.
- Updates injected processes' structures with the goal that the malicious code will be executed.
- Resumes the suspended thread. Now, the malicious code begins executing inside the container made for the legitimate.exe.

The upside of hollow process injection is that the user can't recognize the injected process from the genuine one utilizing ordinary tools. Malware that utilizes process hollowing exclusively or as a malware loader, are Stuxnet and Careto [49], DarkComet [72], Dridex [73], Skeeyah [33].

3.5.6 Thread Execution Hijacking

In this method, an adversary targets an existing thread of a process and avoids any noisy process or thread creation activities. After getting a handle to the target thread, the adversary places the thread into suspended mode by calling SuspendThread API to perform its injection. The attacker calls VirtualAllocEx API and WriteProcessMemory API to allot memory and play out the code injection. The code can contain shellcode, the path to the malevolent DLL, and the address of LoadLibrary. Then, the attacker continues the thread to execute the shellcode that it has written to the host process.

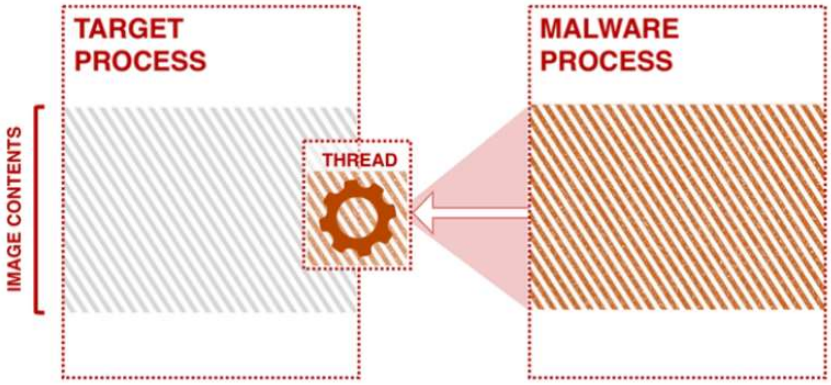


FIGURE 3.11 Thread execution hijacking [35].

3.5.7 Asynchronous Procedure Call (APC) Injection

APC injection is generally performed by joining malicious code to the APC Queue of a process's thread. Queued APC functions are executed when the thread enters an alterable state [74]. This injection makes use of QueueUserAPC API to begin a thread in the target process after writing the shellcode to its memory. The malware can utilize Asynchronous Procedure Calls (APC) to force another thread to execute their custom code by joining it to the APC Queue of the target thread. Each thread has a queue of APCs which are waiting for execution upon the target thread entering an alterable state. An adversary mostly looks for any thread that is in an alterable state, and afterwards calls OpenThread API and QueueUserAPC API to queue an APC to a thread. A malware that utilizes this method is Almanahe.

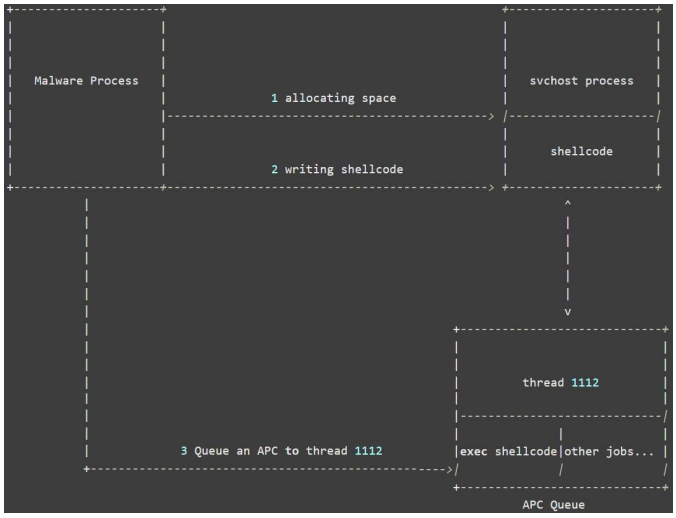


FIGURE 3.12 APC injection [75].

3.5.8 AtomBombing

This method utilizes windows atom tables for writing into the memory of another process. It additionally depends on APC injection. AtomBombing may influence all Windows versions.

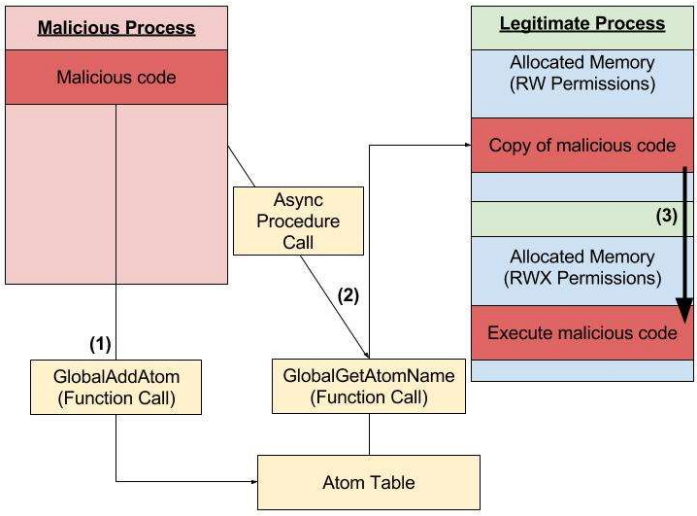


FIGURE 3.13 AtomBombing injection [76].

AtomBombing works in three principle stages:

- Write-What-Where: Writing discretionary data to arbitrary locations in the target process's address space.
- Execution: Capturing a thread of the target process to execute the code that is written in stage one.
- Restoration: Cleaning up and reestablishing the execution of the thread captured in stage two.

3.6 Threat Model and Assumptions

In this section, we characterize the extent of our work. In this work, we explore various process injection methods. This work centers around the identification of eight distinct executions of process injection: to be specific, Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, Portable Executable Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. We center further around the detection of process injections inside the virtualization environment.

3.6.1 Threat Model

- This work centers around process injection exploit inside the virtualization environment.
- Our study emphasis on Windows VMs because it is as yet a definitive centered operating system by malicious software as detailed in [77] [78] [79] and has a bigger attack surface contrasted with the Linux system.
- Our expected attack environments are virtualization environments, such as those utilizing the Infrastructure as a Service (IaaS) cloud model, where the host operating system has limited-to-no control over its guest system working.
- This work set defensive attempt on a detection approach and the detection granularity is on a memory region basis.

3.6.2 Assumptions

- No inter-VM attacks exist.
- The cloud provider, its administrator, and its infrastructures are trustworthy and a Hypervisor or Virtual Machine Monitor (VMM) is free from harm.
- The empty VAD area (totally comprising of zeros) is viewed as harmless. This assists us with limiting false positives.

3.7 Proposed Detection Model

In this work, we distinguish malicious conduct by utilizing the Volatility framework [80], Kernel-based Virtual Machine (KVM) hypervisor [81], and LibVMI [82]. Fig. 3.14 shows the design of our proposed detection framework. The KVM hypervisor offers types of assistance that permit various computer OSs to execute on the same hardware concurrently.

The KVM working at the Virtual Machine Monitor (VMM) or hypervisor is set up on the host machine. Virtual Machines dispatched by the KVM hypervisor have Windows 7, Windows 8.1 and Windows 10 guest OS running on it. LibVMI is a library that can monitor a running VM's low-level details by viewing its memory. LibVMI python bindings (version-3.4) integrated with the Volatility framework (version-2.6.1) is set up on the host OS. Dynamic malware analysis is performed using the Volatility framework and LibVMI python bindings. Our proposed framework leverages Virtual Machine Introspection (VMI) and memory forensic technology to analyze virtual machine's memory in real-time.

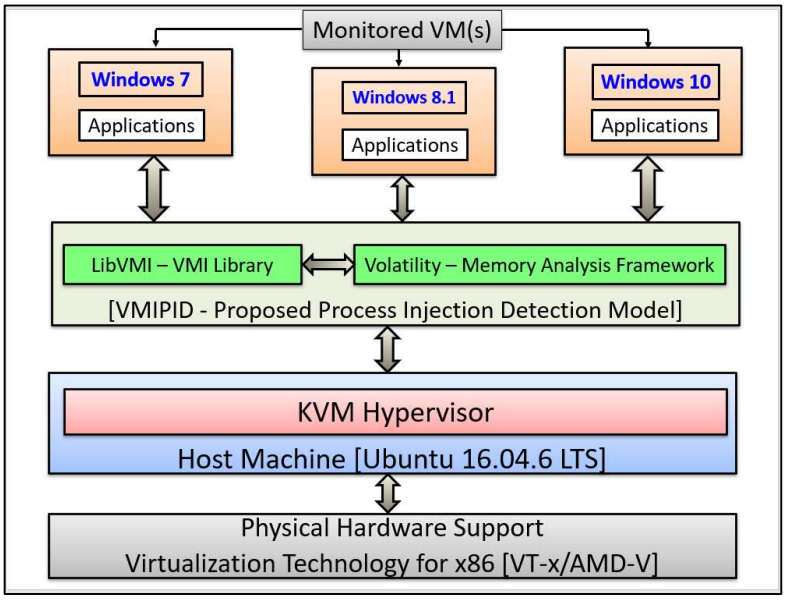


FIGURE 3.14 The architecture of our proposed system.

The malwares are executed on Windows-based VMs in the form of process injection exploitations. After the injection is performed, an actual memory picture of the virtual machine's core is retrieved by creating a dump file with the goal that it tends to be examined later or probably we straightforwardly get to the memory of a running VM utilizing LibVMI python binding and analyze it for a conceivable sign of process injection in real-time. We integrate our proposed framework with the Volatility structure and LibVMI to recognize the VM's memory for high-level analysis. The VMI-based Process Injection Detection (VMIPID) model scans for injected memory regions that are created as a result of process injection. We implement our approach in a plugin for the memory forensic framework Volatility, which automatically reports any memory region containing injected codes, and evaluate it against implementations of different hiding techniques.

3.8 Methodology of Research

There are numerous strategies to inject malignant code into a running process and execute it. Although each process injection strategy varies in execution, affects different memory-resident data structure of the victim process and API calls being used yet all process injection method shares one common property – allocates memory in the address space of the victim process. Most instances of injected code allocate a new memory region (VAD) inside the virtual address space of the victim process. This work proposes a novel method for identifying potentially injected memory areas inside a victim process's virtual address space. The proposed detection approach looks at each memory region of all active processes to conclude if it is the consequence of code injection.

Although there exist different process injection procedures in the wild, this work aims to identify eight distinct executions of process injection: specifically, Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, Portable Executable Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. All these process injection strategies are unnoticeable by a single security product, tool or solution. Rather than simply recognizing infected processes, we mean to identify an actual malicious memory region inside the infected process and thereby pinpoint the malware exactly. Our proposed framework examines injected memory regions that are created as an outcome of code injection. Following algorithms have been designed and executed as a volatility plugin (ProcInjectionsFind) to identify process injection of varied types in virtualized environments.

The steps to find out Suspicious Processes are as per the following.

TABLE 3.1 Determine suspicious processes.

Input: Primary memory of a running VM OR a malware infected memory image.
Output: Suspicious process list containing process-thread id
1: Scan all active processes 2: Enumerate process' handles in each active process 3: Filter process' handles of type THREAD 4: Examine a thread not handled or created by the process under which it executes (*) 5: Append thread' handle PID and TID in the suspicious process list (*) The following thread' handles have been excluded in step 4. • Handles created by csrss.exe • Handles created by its parent process

- csrss.exe is associated with the formation of each process and thread (aside from itself and the processes that started before it) [49].
- A parent process might create the handle of type THREAD in its child process for an authentic reason.

TABLE 3.2 The proposed detection algorithms.

Algorithm 3.1: To detect "Remote DLL injection Via CreateRemoteThread and LoadLibrary"
Input: Primary memory of a running VM OR a malware infected memory image.
Output: Display injected process ID, process name, DLL full name, and the process' corresponding VAD information.
1: Determine suspicious processes from steps listed in TABLE 3.1 2: For each thread in the suspicious process list, perform the following checks. 3: Relate thread with its corresponding VAD → Check for the file mapped on disk → Thread is mapped to kernel32.dll. 4: Scan thread's execution for LoadLibrary (or LoadLibraryEx) API function. 5: Correlate DLLs with thread (that is responsible for injecting malicious DLL) by associating DLL's load time with thread's create time using defined time frame and append it to the list of suspicious DLLs. 6: Check for the DLL entry in the process' IAT, i.e., injected DLL does not have any corresponding entry in the process' IAT. 7: Mark the DLL & respective memory region as suspicious. 8: Dump an entire VAD related to a suspicious memory region. 9: Cross-check with VirusTotal score of the dumped VAD to confirm injection.
Algorithm 3.2: To detect "Thread execution hijacking"
Input: Primary memory of a running VM OR a malware infected memory image.
Output: Display various attributes of each injected memory region.
1: Scan all active processes. 2: Enumerate all threads in each active process. 3: Found thread id in the suspicious process list. 4: Check if a thread is suspended → Thread's State is 'Waiting' and WaitReason is 'Suspended'. 5: Traverse process' VADs and perform the following check on memory regions (VAD). • Any VAD region marked as private, has a VadS tag, and with execute permission. 6: Mark the matching memory region as suspicious. 7: Dump an entire VAD related to a suspicious memory region. 8: Cross-check with VirusTotal score of the dumped VAD to confirm injection.
Algorithm 3.3: To detect the following type of injections.

a. Remote thread injection using CreateRemoteThread	b. PE injection
c. Reflective DLL injection	d. Hollow process injection
e. APC injection	f. AtomBombing

Input: Primary memory of a running VM OR a malware infected memory image.
Output: Display various attributes of each injected memory region.
1: Scan all active processes. 2: Enumerate all threads in each active process. 3: Extract thread's entry point from Win32StartAddress attribute. 4: Apply the following injection filters at the thread's entry point. • Any thread in the process mapped to a VAD without a file object. • Any thread in the process mapped to a VAD with a file object and the memory is committed, but the type of file object is not an IMAGE FILE. • Any thread in the process mapped to a VAD that contains an exe file object that is different from the loaded process's image file. • Any thread in the process mapped to a VAD that contains an exe file object that is same as loaded process's image file, but the thread is suspended, i.e., thread's State is 'Waiting' and WaitReason is 'Suspended'. 5: Examine processes VADs. 6: Apply following injection filters at VAD region. • Any VAD region representing a memory mapped file (type _MMVAD (Vad) or _MMVAD_LONG (VadL)), but the fields VadImageMap in Vad Type and Image in Control Flag are not set. • Any VAD region which is marked as private, committed, memory-resident, has a VadS tag, with execute permission, and the type is VadNone. 7: Dump an entire VAD related to a suspicious memory region. 8: Cross-check with VirusTotal score of the dumped VAD to confirm injection.

The proposed algorithm analyzes each running process's thread and memory segment to decide whether it is the aftereffect of process injection. All the steps of the proposed detection algorithms listed in Table 3.1 and Table 3.2 are converted into a single volatility plugin/module known as ProcInjectionsFind, which can be called from Volatility's command prompt. This module runs a few checks to pinpoint malicious/injected memory sections and prints various attributes of each injected memory area that match our rules characterized in the above algorithms. A volatility plugin ProcInjectionsFind runs against the Windows memory image or memory of a live VM and scans each process's thread and memory area to conclude if it is the result of process injection. The steps mentioned above are carried out on malware infected memory images and memory of live VMs and the outcomes are confirmed.

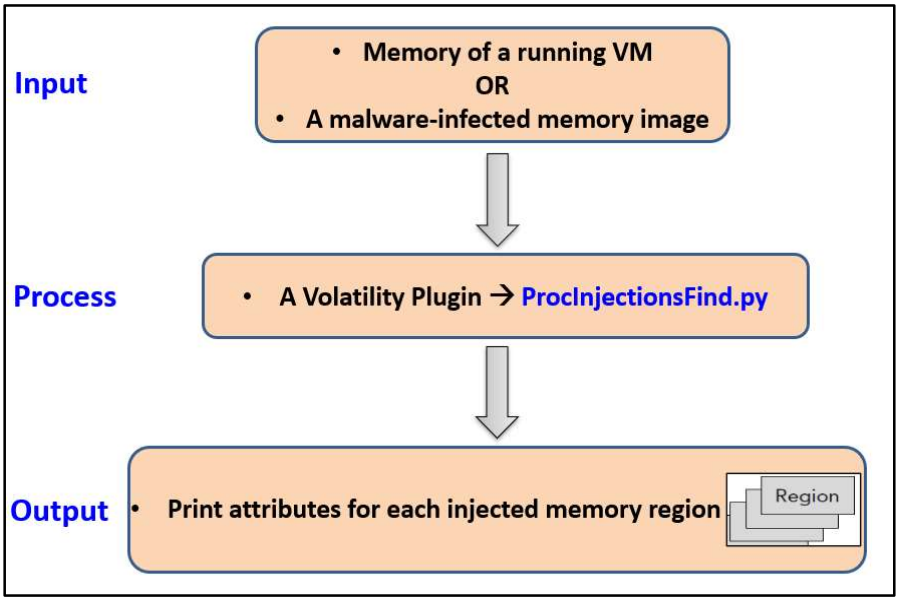


FIGURE 3.15 Proposed system workflow.

Figure 3.15 shows our proposed framework workflow. It comprises three phases: namely, input, process, and output. The framework accepts the live memory of a running VM or a malware-contaminated memory image as an input. The ProcInjectionsFind module works on the input provided in the first stage and shows different attributes of each injected memory region as an output.

3.9 Malware Hiding Technique (Bypassing Common Detection Method) Covered in This Work

The VAD's protection field just holds back the initial protection, set during its allocation. The aggressor simply needs to allocate the memory at first with protection without the WRITE or EXECUTE right and later on, add this right to the memory section containing the malicious code. Malevolent executables have been altered to assign the memory section with READONLY protection at first and later change it to EXECUTE_READWRITE. The VirtualProtectEx API function changes the protection on a memory section in the virtual address space of a stated process [83]. One such example is to VirtualAllocEx a block of virtual memory for read/write access at first, populate it with executable instructions, and afterwards VirtualProtectEx it a while later to eliminate write access and give it execution access.

3.10 Experimental Environment Setup

In this part, a brief explanation of our experimental setup is described. We carry out experiments on the host machine, which had the configurations shown in Table 3.3. To set up our virtualization environment, we install and configure KVM on Ubuntu 16.04.6 LTS. We select the KVM hypervisor in our experimental setup because according to the review directed by Kourai and Nakamura [84] where they looked at the performance of KVM and Xen hypervisor. Based on their experiments, they inferred that KVM was the more proficient hypervisor when performing VMI. KVM is also one of the supported hypervisors by Openstack. Openstack is an open-source distributed computing software platform that is utilized basically to host IaaS [85]. There are at present more than 670 organizations utilizing Openstack cloud computing platforms.

TABLE 3.3 Testbed configurations.

Host OS	Ubuntu 16.04.6 LTS
Host OS Type	64-bit
Linux Kernel	Linux 4.15.0-74-generic
Architecture	X86_64
Processor	Intel(R) Core™ i5-8265U CPU @ 1.60GHz x 8
Disk	1 TB
Number of cores & threads	4 & 8
Physical memory (RAM)	8 GB
Hypervisor (VMM)	KVM
Virtual Machine – 1	OS – Windows 7, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Virtual Machine – 2	OS – Windows 8.1, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Virtual Machine – 3	OS – Windows 10, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Tools / Framework used	LibVMI python bindings (version-3.4) & Volatility framework (version-2.6.1) (Both are open-source tools)

We utilize the Infrastructure as a Service (IaaS) cloud model as an anticipated attack environment. The reasons for selecting Linux as the host OS are manifold. Numerous cloud platforms make use of Linux as host OS. According to the Linux Foundation, nine of the top ten public clouds run on Linux. Linux is running high percentage of the public cloud workload. Linux offers low total cost of ownership, high flexibility, scalability, security and reliability. Linux is a free and open-source operating system. We have not picked windows

as the host OS because we focused on the aspects of cost, the functionality provided, hardware compatibility, support, reliability, security, pre-built software, and cloud-readiness.

We set up three virtual machines by installing Windows 7, Windows 8.1 and Windows 10 as guest OS. The reasons for preferring three versions of windows in VMs are numerous. Each of these three versions of windows are the most generally utilized Windows platforms. Windows operating systems are the most vulnerable to malware attacks and are targeted more than any other operating system. Computers that run Windows—the world's most popular operating system—are especially prone to malware attacks. Most hypervisors support VMs running the Windows OS as a guest.

We leverage open-source tools: in particular, LibVMI python binding and Volatility framework for our experimentation. We utilized LibVMI as a VMI tool and a Volatility system as memory investigation instruments to perform dynamic malware examination and to break down more significant level semantic information from live memory data in VMs.

CHAPTER – 4

4. Results and Discussion

4.1 Evaluation of A Proposed Model

The proposed VMIPID model reads the virtual machine's memory and scans each running process's memory sections for the presence of injected code in real-time. The model classifies each of these memory regions as being either harmless or malevolent. The proposed model runs several checks to spot malicious/injected memory areas.

4.2 Evaluation Metrics

To assess the performance of the proposed VMIPID model, we utilized several assessment metrics. These include Precision (P), Recall (R), False Positive Rate (FPR), Detection Rate, Accuracy, and F1-Score. The definition of measures utilized in assessment metrics can be seen in Table 4.1.

TABLE 4.1 Definition of measures used in evaluation metrics.

Measures	Definition
True Positive (TP)	The number of correctly identified injected memory regions.
False Positive (FP)	The number of incorrectly identified injected memory regions.
True Negative (TN)	The number of correctly identified benign memory regions.
False Negative (FN)	The number of incorrectly identified benign memory regions.

We require some performance evaluation metrics to discover how well our proposed model fits for the classification of a memory region. In the following subsections, we highlight different performance assessment metrics used in this work such as precision, recall [86], false positive rate, accuracy and F1-score [87].

4.2.1 Precision (P)

Precision refers to "what number of chosen data items are significant". In various implications, how many of them are positive, from the observations that an algorithm has expected to be positive. It depicts a positive predictive value. According to the equation addressed in (4.1), precision is equivalent to the total number of true positive (TP) divided by the total of true positive (TP) and false positive (FP):

$$\text{Precision} = \frac{\mathbf{TP}}{\mathbf{TP + FP}} \tag{4.1}$$

4.2.2 Recall (R) / True Positive Rate / Completeness / Sensitivity

A recall is otherwise called true positive rate or completeness. Recall essentially signifies "what number of significant data items are chosen". Actually, out of all positive observations, how many of them are predicted correctly by the algorithm. Both precision and recall are based on relevance. The detection framework is treated to be complete when all injected memory areas across multiple targeted processes are being discovered. As per the equation addressed in (4.2), recall is identical to the total number of true positive (TP) divided by the total of true positive (TP) and false negative (FN):

$$\text{Recall} = \frac{\mathbf{TP}}{\mathbf{TP + FN}} \tag{4.2}$$

4.2.3 False Positive Rate (FPR)

In specialized terms, the false positive rate is characterized as the likelihood of falsely rejecting the null hypothesis. In basic words, a false positive implies that the truth is negative, however, the test predicts a positive [88]. It's the likelihood that a false alarm will be raised: that a positive outcome will be given when the true value is negative. According to the equation addressed in (4.3), the false positive rate is identical to the total number of false positive (FP) divided by the total of false positive (FP) and true negative (TN):

$$\text{False Positive Rate} = \frac{\mathbf{FP}}{\mathbf{FP + TN}} \tag{4.3}$$

4.2.4 Accuracy

It is the most generally utilized and conceivably the best option for assessing the output of an algorithm in classification problems. It is calculated as a total number of accurately classified samples to the total number of samples in the dataset as per the equation (4.4). To assess the performance of the proposed model, we can characterize accuracy as correctly distinguished injected memory regions by a total number of memory regions on the memory image.

$$\text{Accuracy} = \frac{\mathbf{TP + TN}}{\mathbf{TP + FP + TN + FN}} \tag{4.4}$$

4.2.5 F1-Score / F-Score / F-Measure

F1-score is otherwise called f-score or f-measure. It is a measure of a test's accuracy. It considers both precision and recall to calculate an algorithm's efficiency. It is the weighted harmonic mean of precision and recall. The mathematical equation for F1-score is addressed using equation (4.5) as follows:

$$\text{F1-Score} = \frac{\mathbf{2 * (Precision * Recall)}}{\mathbf{Precision + Recall}} \tag{4.5}$$

4.2.6 Detection Rate

The detection rate is one of the significant performance measurements to assess the effectiveness of the proposed algorithm. It addresses the proportion of true positive samples and the total number of samples in the review. The mathematical formula for detection rate is addressed using equation (4.6) as follows:

$$\text{Detection Rate} = \frac{\mathbf{TP}}{\mathbf{(TP+TN+FP+FN)}} \tag{4.6}$$

4.3 Results and Discussion

In this section, we check the performance of our proposed model (VMIPID) by several experiments. We discuss the performance measurement of the proposed framework with various assessment metrics. Finally, we will compare the proposed approach with all the conventional approaches available in the literature.

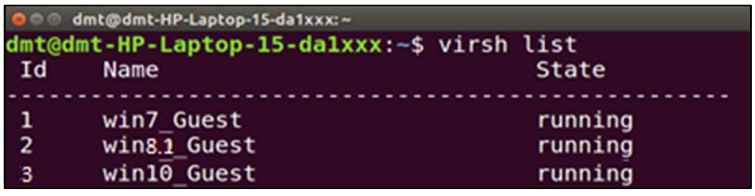


FIGURE 4.1 List of running VMs on the host.

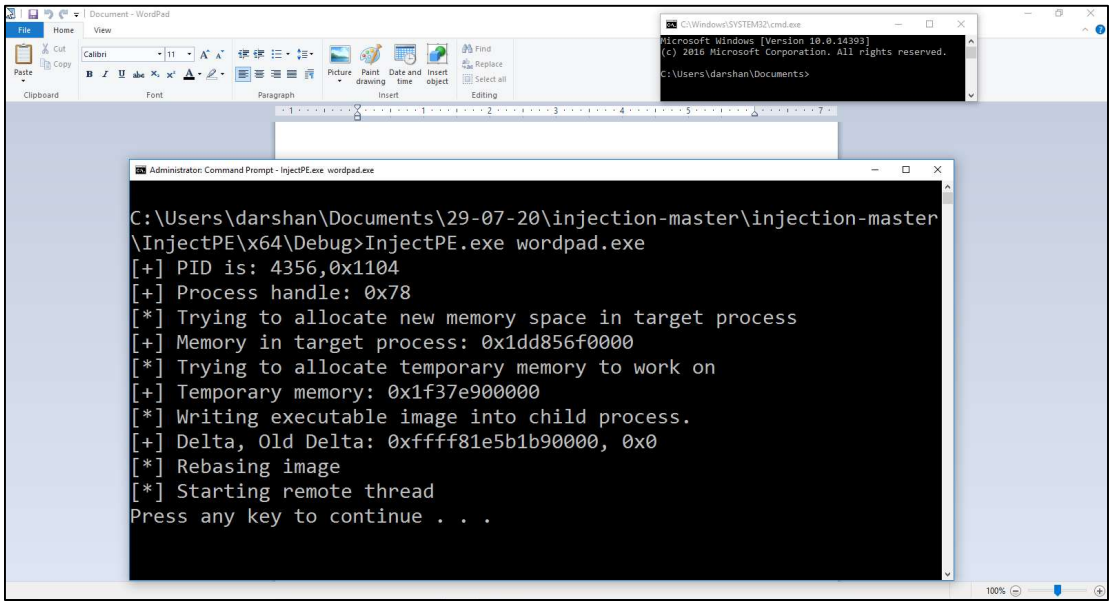


FIGURE 4.2 Performing PE injection on win10_VM.

Fig. 4.1 depicts a list of live virtual machines on the host. Fig. 4.2 shows how the PE injection is performed on the Windows 10 VM using the proof of concepts referenced in TABLE 4.2. At first, wordpad.exe (target/victim process) is opened, which is used as a reference for the injected instance. The command for the injection is shown in Fig. 4.2.

FIGURE 4.3 Acquiring memory image of live win10_VM.

After performing PE injection, we gained a memory picture of win10_VM utilizing the 'virsh dump' command as shown in Fig. 4.3. The ProcInjectionsFind volatility module is designed to automate the identification of different process injection strategies which are described in this work. We can run the ProcInjectionsFind plugin on a malware infected (PE injection) memory image as shown in Fig. 4.4 or memory of virtual machine in real-time as shown in Fig. 4.5.

```
(venv) root@dmt-HP-Laptop-15-da1xxx:/home/dmt/volatility# python vol.py --plugins=/home/dmt/volatility/procinjectionsfind/
 -f /home/dmt/memory-dump-files/win10-Guest-clone.mem --profile=Win10x64_14393 procinjectionsfind
Volatility Foundation Volatility Framework 2.6.1


Process Injections Find Information:

------------------------------------------------------------------------------
Process: wordpad.exe    PID: 4356       PPID: 2832      Active Threads: 4
------------------------------------------------------------------------------
VAD Info:
        VAD Base Address: 0x1dd856f0000
        VAD End Address: 0x1dd85717fff
        VAD Size: 0x27fff
        VAD Tag: VadS
        VAD Protection: PAGE_EXECUTE_READWRITE
        VAD Flags: PrivateMemory: 1, Protection: 6
        VAD Type: VadNone
        VAD Mapped File: ''

Disassembly Info:
        0x1dd856f0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
        0x1dd856f0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
        0x1dd856f0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
        0x1dd856f0030  00 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00   ................

        0x856f0000 4d               DEC EBP
        0x856f0001 5a               POP EDX
        0x856f0002 90               NOP
        0x856f0003 0003             ADD [EBX], AL
        0x856f0005 0000             ADD [EAX], AL
        0x856f0007 000400           ADD [EAX+EAX], AL
        0x856f000a 0000             ADD [EAX], AL
        0x856f000c ff               DB 0xff
        0x856f000d ff00             INC DWORD [EAX]
        0x856f000f 00b800000000     ADD [EAX+0x0], BH
        0x856f0015 0000             ADD [EAX], AL
        0x856f0017 004000           ADD [EAX+0x0], AL
        0x856f001a 0000             ADD [EAX], AL
        0x856f001c 0000             ADD [EAX], AL
        0x856f001e 0000             ADD [EAX], AL
        0x856f0020 0000             ADD [EAX], AL
        0x856f0022 0000             ADD [EAX], AL
        0x856f0024 0000             ADD [EAX], AL
        0x856f0026 0000             ADD [EAX], AL
        0x856f0028 0000             ADD [EAX], AL
        0x856f002a 0000             ADD [EAX], AL
        0x856f002c 0000             ADD [EAX], AL
        0x856f002e 0000             ADD [EAX], AL
        0x856f0030 0000             ADD [EAX], AL
        0x856f0032 0000             ADD [EAX], AL
        0x856f0034 0000             ADD [EAX], AL
        0x856f0036 0000             ADD [EAX], AL
        0x856f0038 0000             ADD [EAX], AL
        0x856f003a 0000             ADD [EAX], AL
        0x856f003c f00000           LOCK ADD [EAX], AL
        0x856f003f 00               DB 0x0

Elapsed Wall-Clock Time: 88.9538369179 seconds

(venv) root@dmt-HP-Laptop-15-da1xxx:/home/dmt/volatility#
```

FIGURE 4.4 Execution of procinjectionsfind plugin on malware (PE injection) infected memory image (win10_VM).

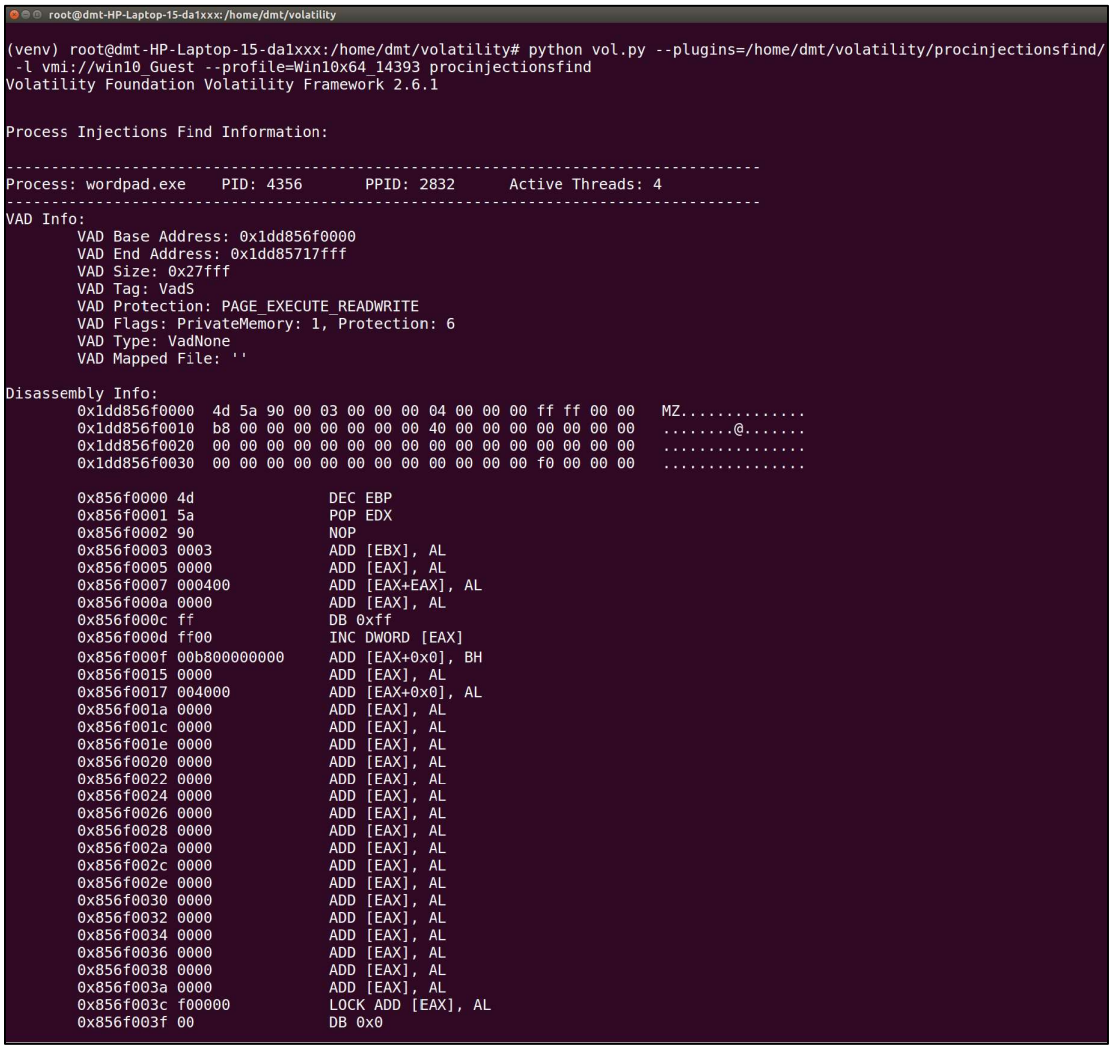


FIGURE 4.5 Execution of procinjectionsfind plugin on the memory of live win10_VM.

The ProcInjectionsFind plugin prints information about the injected/victim process's VADs, disassembly, and a hex-dump at the VADs base address. We can likewise dump injected memory region to disk with the ProcInjectionsFind plugin for later investigation or to confirm the derived outcomes.

TABLE 4.2 Evaluation with process injection Proof of Concepts (PoCs).

Sr No	Process Injection Techniques	PoCs Used
1	Remote DLL injection	• Injection techniques. GitHub – theevilbit/injection [97]. • Process Injection Techniques. GitHub – secrary/InjectProc [101]. • Seven different DLL injection techniques. GitHub – fdiskyou/injectAllTheThings [100]. • Windows Process Injection. GitHub - CptGibbon/Windows-Process-Injection: Some simple process injection techniques targeting the Windows platform [103].
2	Remote thread injection	• Injection techniques. GitHub – theevilbit/injection [97].
3	PE injection	• Injection techniques. GitHub – theevilbit/injection [97]. • Windows Process Injection. GitHub - CptGibbon/Windows-Process-Injection: Some simple process injection techniques targeting the Windows platform [103]. • Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
4	Reflective DLL injection	• Reflective DLL Injection. GitHub – stephenfewer/ReflectiveDLLInjection [98] • Seven different DLL injection techniques. GitHub – fdiskyou/injectAllTheThings [100]. • Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
5	Hollow process injection	• Process Hollowing. GitHub – m0n0ph1/Process-Hollowing [99] • Injection techniques. GitHub – theevilbit/injection [97]. • Windows Process Injection. GitHub - CptGibbon/Windows-Process-Injection: Some simple process injection techniques targeting the Windows platform [103]. • Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
6	Thread execution hijacking	• Injection techniques. GitHub – theevilbit/injection [97]. • Windows Process Injection. GitHub - CptGibbon/Windows-Process-Injection: Some simple process injection techniques targeting the Windows platform [103].
7	APC injection	• Injection techniques. GitHub – theevilbit/injection [97]. • Process Injection Techniques. GitHub – secrary/InjectProc [101].
8	AtomBombing	• AtomBombing. GitHub - BreakingMalwareResearch/atom-bombing: Brand New Code Injection for Windows [102]. • Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
Malware Hiding Technique PoCs		
1	PE injection	• Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
2	Reflective DLL injection	• Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104]
3	Hollow process injection	• Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub [104] • KSLSample.vmem, Process hollowing in different approaches [93]

To perform process injection, the source code of various malware projects has been utilized as shown in Table 4.2. The source code is compiled using Microsoft Visual Studio Community 2017, Version 15.9.37 wherever applicable. After the injection, we procured memory images of malware infected Windows VMs.

4.3.1 Comparison of Evaluation Metrics with Existing Approaches

We evaluate our results in contrast to the malware hiding technique referenced in section 3.9. We used KSLSample.vmem [93] memory dump in our experimentation that contains different variations of process hollowing. The memory dump (KSLSample.vmem) contains three different variations of the process hollowing and holds five processes (svchost.exe) that were liable to process hollowing in various ways. All these five processes were effectively revealed by our executed module (procinjectionsfind).

TABLE 4.3 Comparison of process injection detection techniques with existing approaches.

Sr No	Process Injection Techniques	Compared With
1	Remote thread injection, PE injection, Reflective DLL injection, Thread execution hijacking, APC injection, and AtomBombing	Malfind [89]
2	Hollow process injection	Hollowfind [90], Malfind [89], Threadmap [91], Malfofind [92]
3	Remote DLL injection	FindDLLInj [70]

TABLE 4.4 Volatility's plugins used in this work.

Volatility Plugins	Description
Malfind [89]	Find hidden or injected code in user mode memory.
Hollowfind [90]	Detect various types of process hollowing techniques.
Threadmap [91]	Detect any user mode memory manipulation and unattended kernel manipulations. Use _ETHREAD structure information to relate a thread with a VAD.
Malfofind [92]	Find code injected using the "Process Hollowing" technique.
Vadinfo [94]	Display extended information about a process's VAD nodes.
Impscan [95]	Scan for calls to API functions.
Volshell [96]	Shell in the memory image. Provide a platform to interactively explore a memory image from a python command prompt.
Python Script (called from Volatility's volshell plugin)	
FindDLLInj [70]	Detect remote DLL injection.

A comparison of different process injection detection techniques with existing methodologies is presented in Table 4.3. We utilized different volatility commands like Malfind, Hollowfind, Threadmap, Malfofind, Vadinfo, Impscan, and Volshell to hunt malware in windows memory images as depicted in Fig. 4.6.

Virtual Machine	Average Performance												Average Sample Size
	Precision (P)	Recall (R)	False Positive Rate (%)	Detection Rate (%)	F1-Score (%)	Accuracy (%)	Precision (P)	Recall (R)	False Positive Rate (%)	Detection Rate (%)	F1-Score (%)	Accuracy (%)	
	Malfind						ProcInjectionsFind						
win7_Guest	0.08	0.91	0.29%	0.01%	12.88%	99.70%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8971
win8.1_Guest	0.06	0.83	0.24%	0.01%	12.80%	99.76%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6474
win10_Guest	0.10	0.75	0.12%	0.01%	22.61%	99.87%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	11439
	Hollowfind						ProcInjectionsFind						
win7_Guest	0.19	1.00	0.09%	0.02%	30.26%	99.91%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.08	1.00	0.19%	0.02%	14.29%	99.81%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.02	1.00	0.29%	0.01%	4.65%	99.71%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Malfind						ProcInjectionsFind						
win7_Guest	0.13	1.00	0.25%	0.02%	21.92%	99.75%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.08	1.00	0.19%	0.02%	14.29%	99.81%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.02	1.00	0.28%	0.01%	4.76%	99.72%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Threadmap						ProcInjectionsFind						
win7_Guest	0.05	1.00	0.28%	0.02%	10.00%	99.72%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.06	1.00	0.28%	0.02%	10.53%	99.72%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.03	1.00	0.21%	0.01%	6.25%	99.79%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Malfofind						ProcInjectionsFind						
win7_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	1.00	1.00	0.00%	0.01%	100.00%	100.00%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	FindDLLInj						ProcInjectionsFind						
win7_Guest	##.##	0.00	0.00%	0.00%	##.##	99.99%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	10582
win8.1_Guest	##.##	0.00	0.00%	0.00%	##.##	99.98%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	5919
win10_Guest	1.00	0.83	0.00%	0.01%	88.89%	100.00%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	13477
	With Malware Hiding Technique												
	Malfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.41%	0.00%	##.##	99.58%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	8631
win8.1_Guest	0.00	0.00	0.28%	0.00%	##.##	99.71%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	12787
win10_Guest	0.00	0.00	0.20%	0.00%	##.##	99.80%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14059
	Hollowfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.87%	0.00%	##.##	99.11%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.00	0.00	0.59%	0.00%	##.##	99.39%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	0.00	0.07%	0.00%	##.##	99.93%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Malfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.10%	0.00%	##.##	99.88%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.00	0.00	0.19%	0.00%	##.##	99.80%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	0.00	0.06%	0.00%	##.##	99.94%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Threadmap						ProcInjectionsFind						
win7_Guest	0.06	1.00	0.39%	0.02%	11.11%	99.61%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.06	1.00	0.27%	0.02%	10.53%	99.73%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	1.00	0.71%	0.00%	0.82%	99.29%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Malfofind						ProcInjectionsFind						
win7_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.50	1.00	0.00%	0.00%	66.67%	100.00%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Note: ##.## indicates undefined (Division by zero)												

FIGURE 4.6 Comparison of evaluation metrics with detection approaches.

We run our module (ProcInjectionsFind) on a total of 75 distinct malware infected memory images (acquired 25 images of each of three VM) depicted in Fig. 4.6, demonstrating that the module works satisfactory.

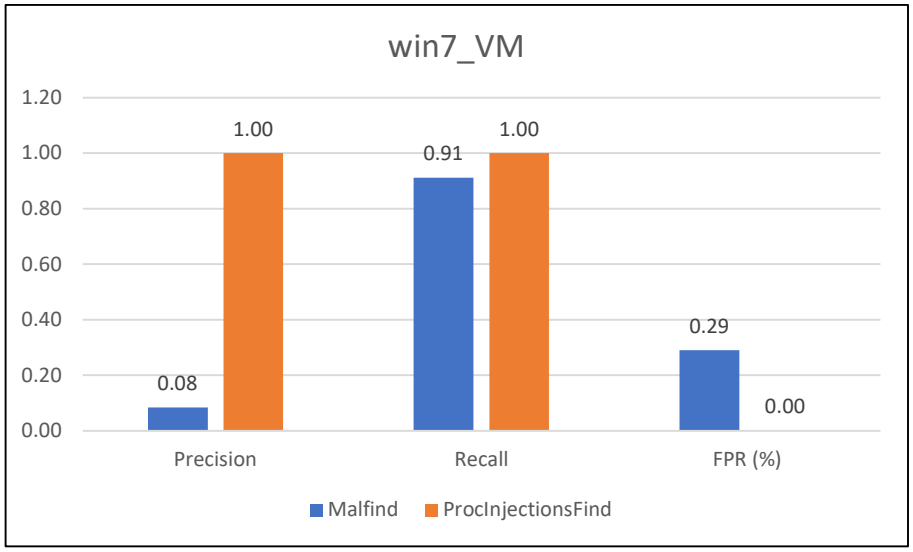


FIGURE 4.7 Comparison of evaluation metrics on win7_VM (Malfind Vs ProcInjectionsFind).

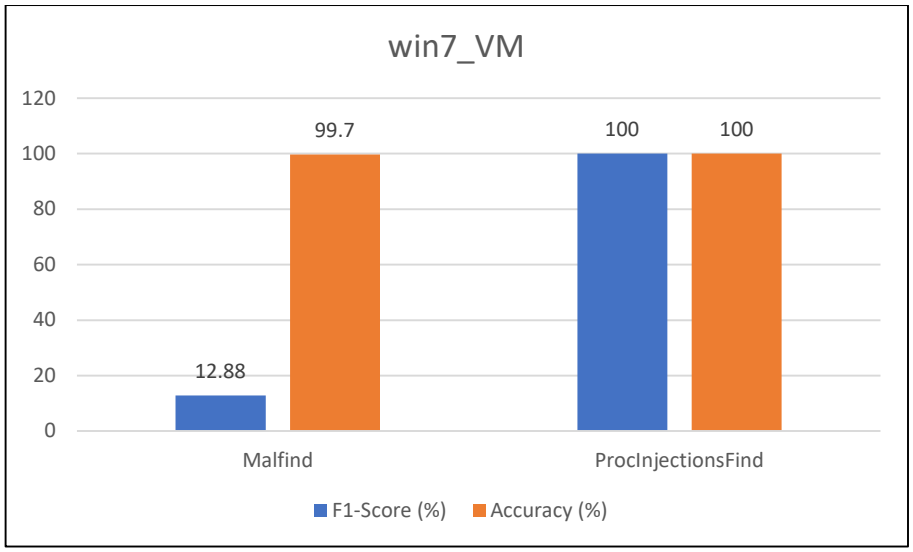


FIGURE 4.8 Comparison of evaluation metrics on win7_VM (Malfind Vs ProcInjectionsFind).

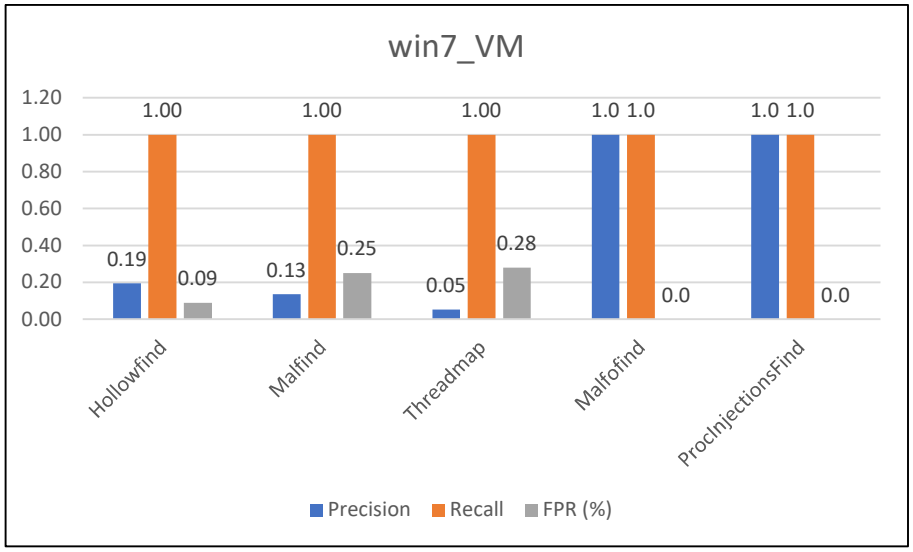


FIGURE 4.9 Comparison of evaluation metrics on win7_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

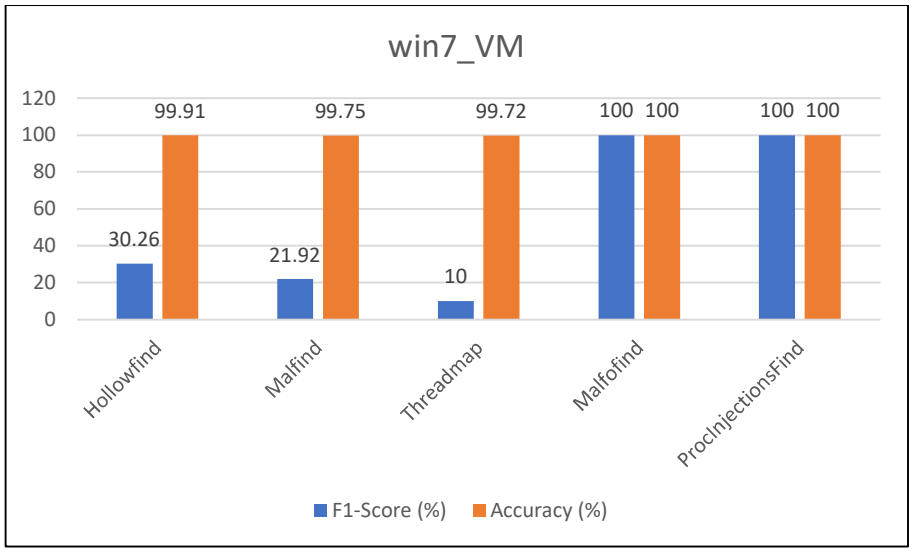


FIGURE 4.10 Comparison of evaluation metrics on win7_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

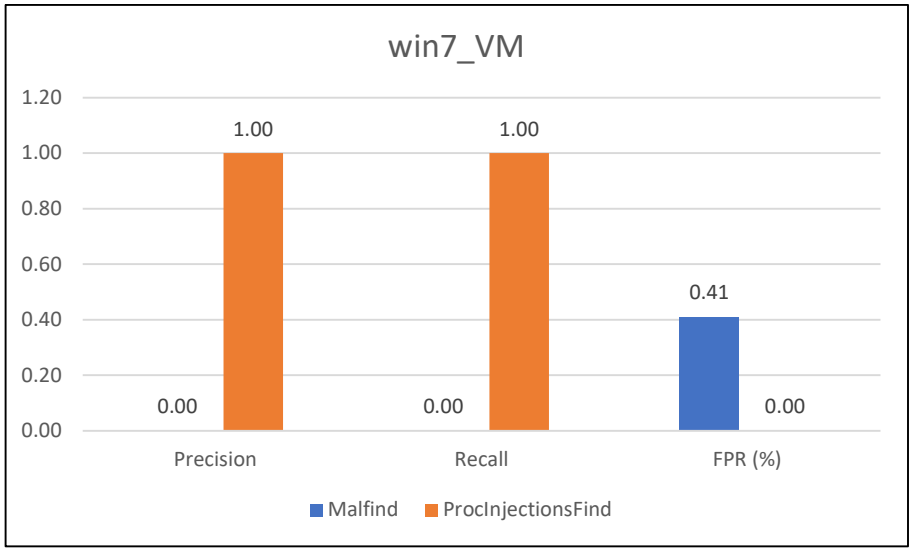


FIGURE 4.11 Comparison of evaluation metrics on win7_VM employing malware hiding technique (Malfind Vs ProcInjectionsFind).

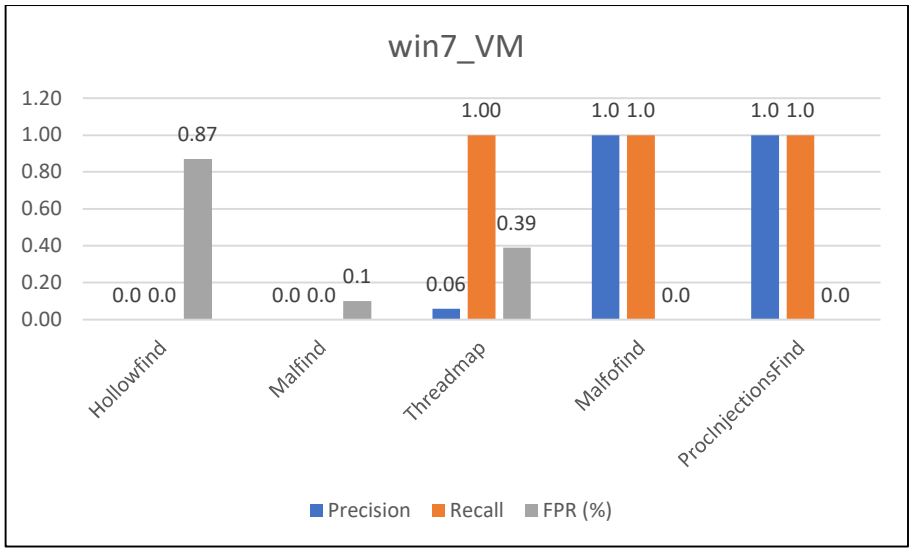


FIGURE 4.12 Comparison of evaluation metrics on Win7_VM employing malware hiding technique (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

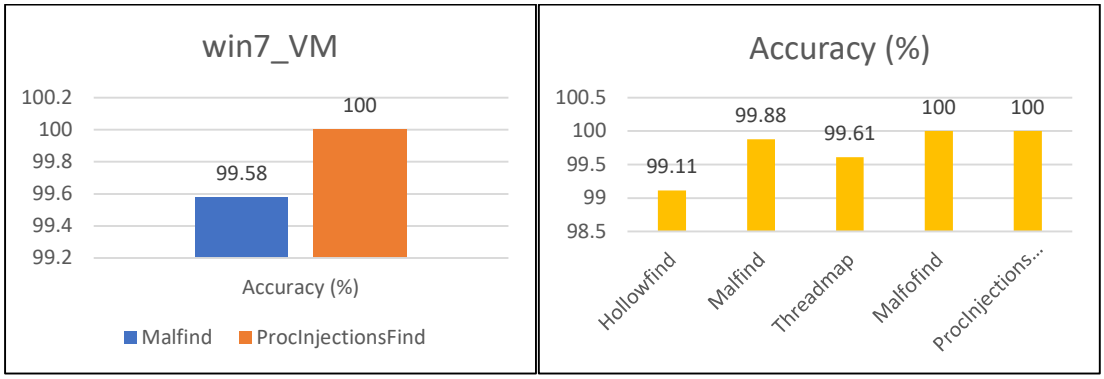


FIGURE 4.13 Comparison of accuracy on Win7_VM employing malware hiding technique.

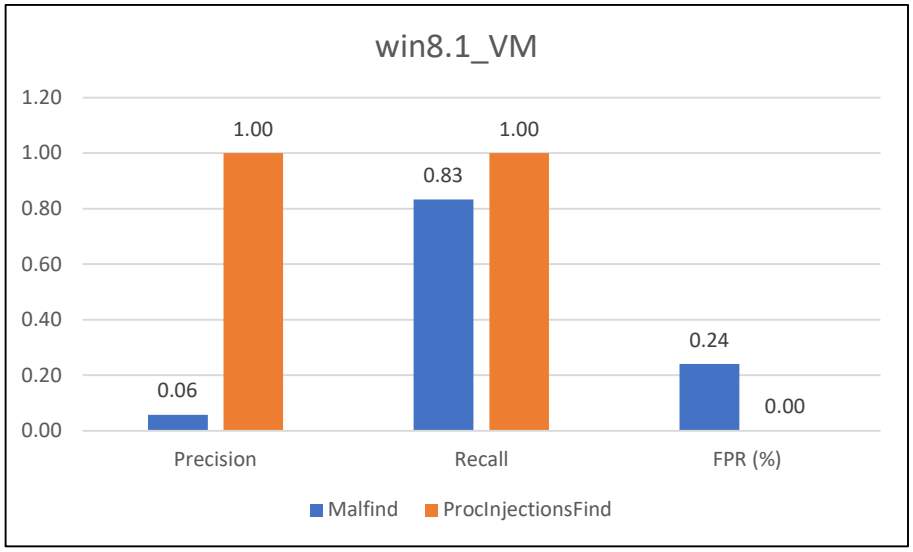


FIGURE 4.14 Comparison of evaluation metrics on win8.1_VM (Malfind Vs ProcInjectionsFind).

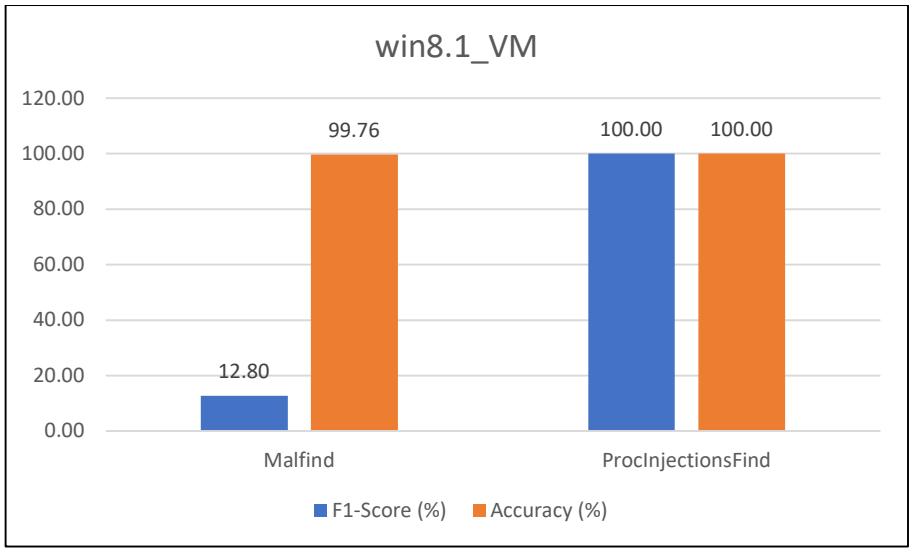


FIGURE 4.15 Comparison of evaluation metrics on win8.1_VM (Malfind Vs ProcInjectionsFind).

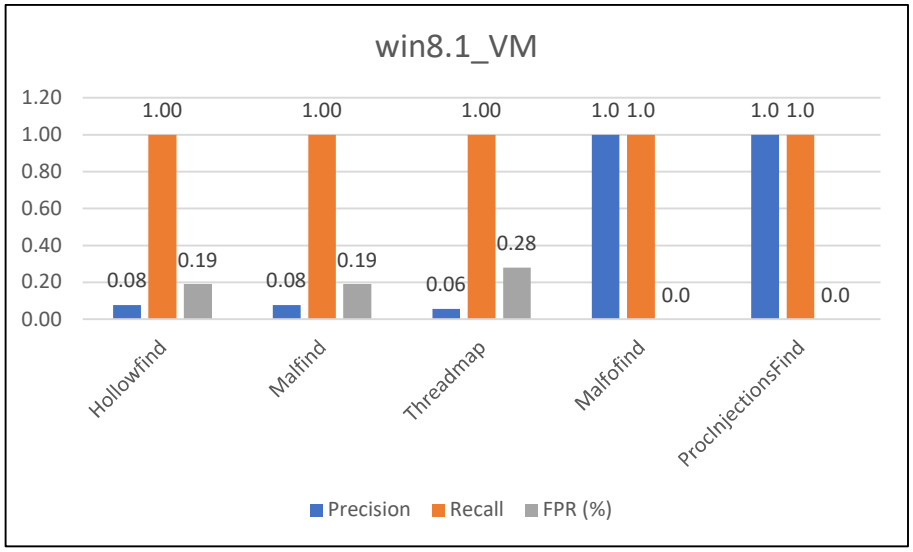


FIGURE 4.16 Comparison of evaluation metrics on win8.1_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

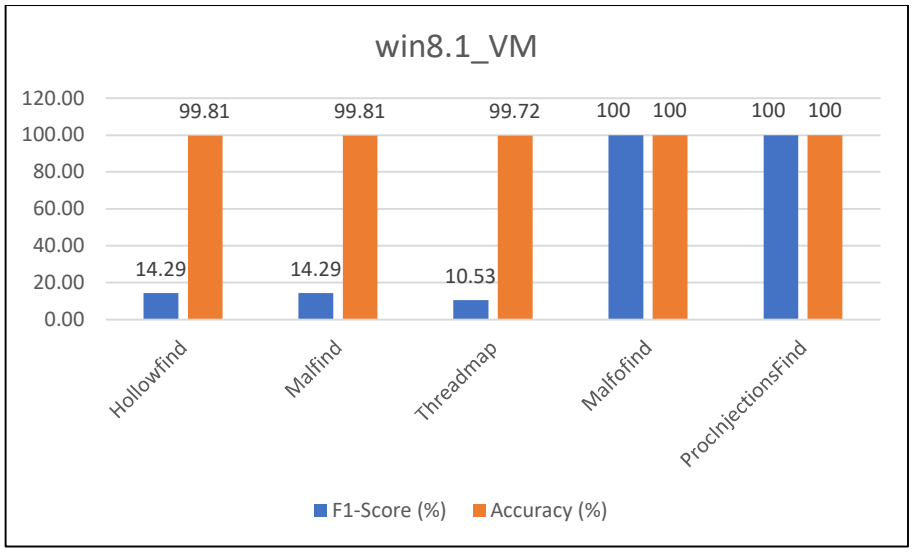


FIGURE 4.17 Comparison of evaluation metrics on win8.1_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

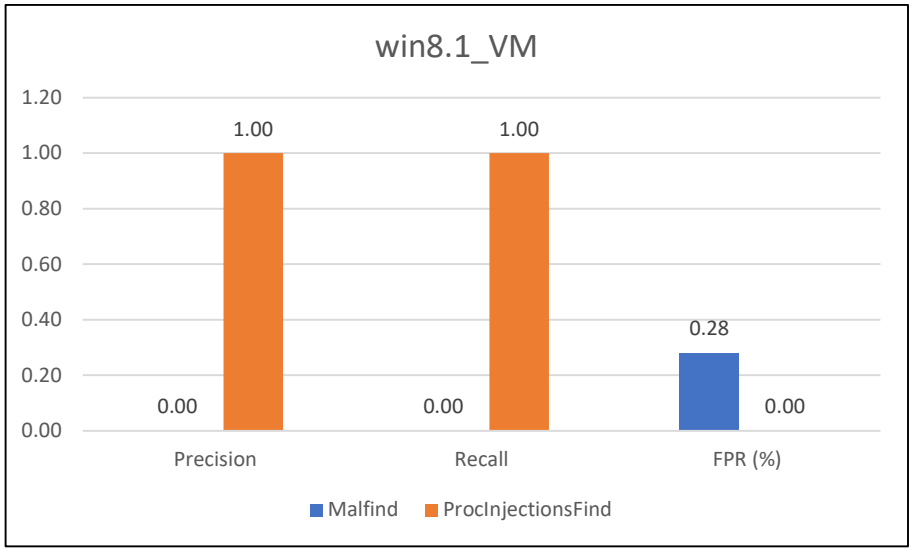


FIGURE 4.18 Comparison of evaluation metrics on win8.1_VM employing malware hiding technique (Malfind Vs ProcInjectionsFind).

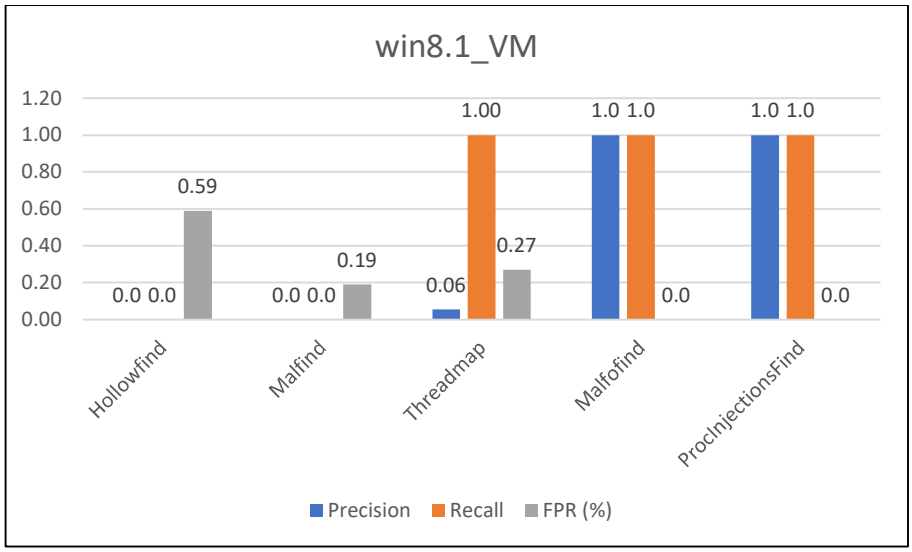


FIGURE 4.19 Comparison of evaluation metrics on win8.1_VM employing malware hiding technique (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

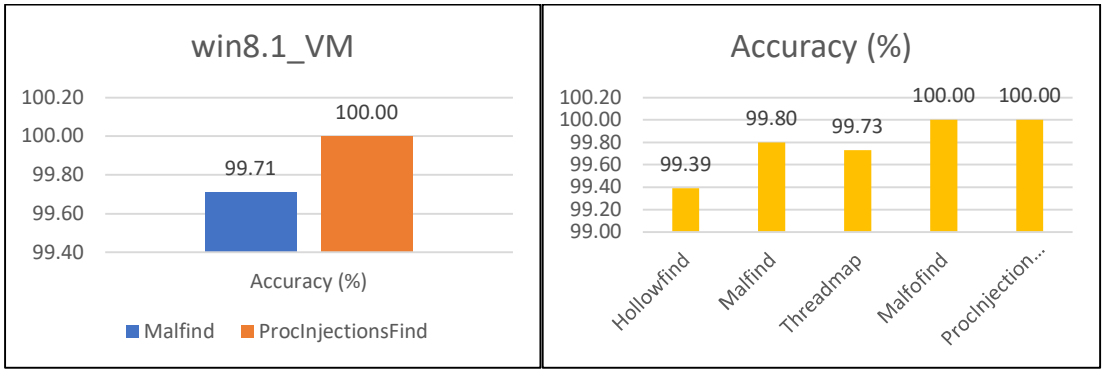


FIGURE 4.20 Comparison of accuracy on win8.1_VM employing malware hiding technique.

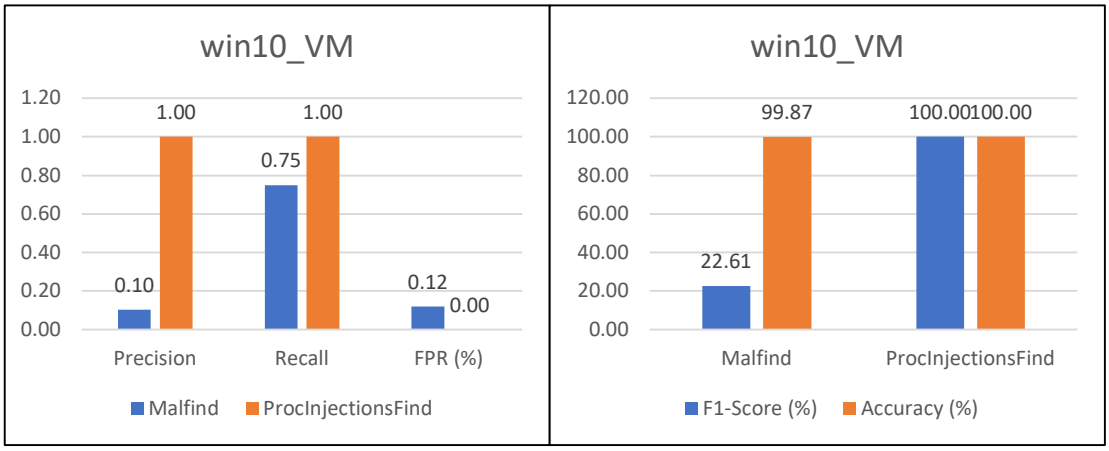


FIGURE 4.21 Comparison of evaluation metrics on win10_VM (Malfind Vs ProcInjectionsFind).

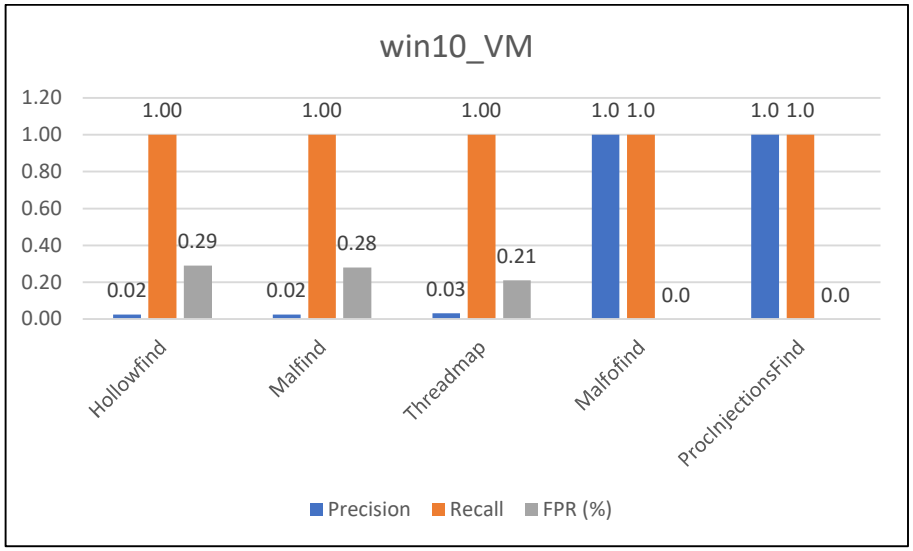


FIGURE 4.22 Comparison of evaluation metrics on win10_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

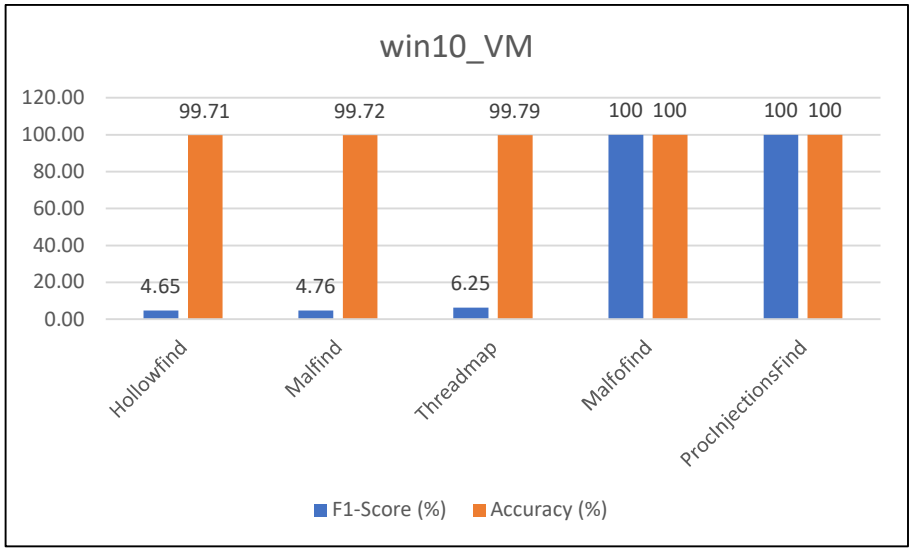


FIGURE 4.23 Comparison of evaluation metrics on win10_VM (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

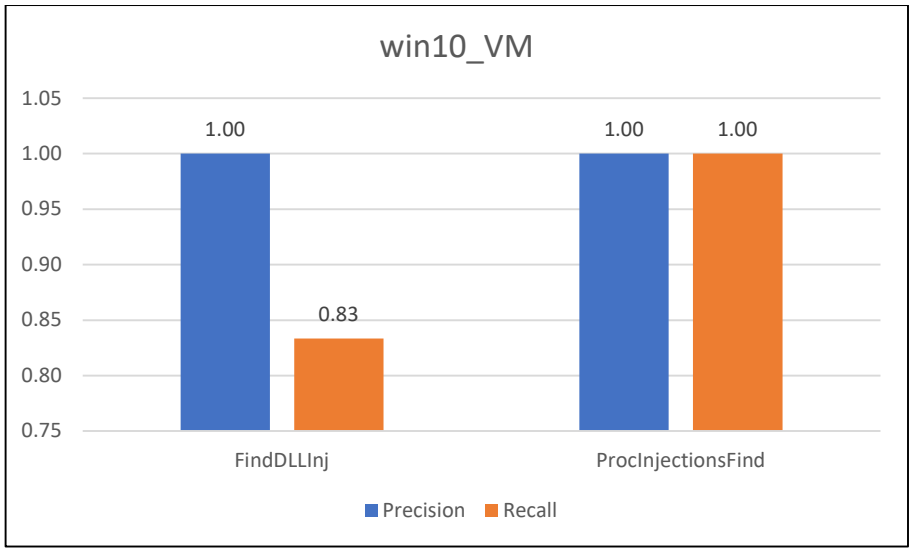


FIGURE 4.24 Comparison of evaluation metrics on win10_VM (FindDLLInj Vs ProcInjectionsFind).

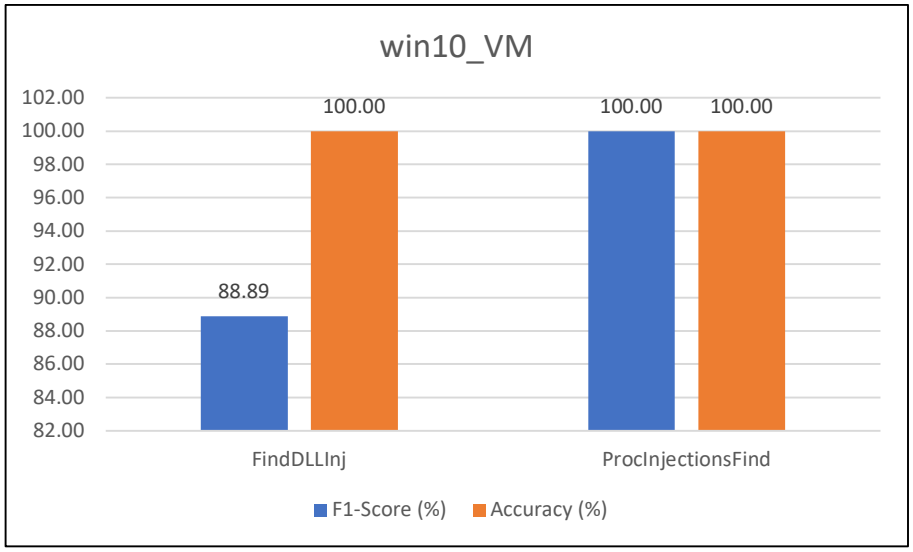


FIGURE 4.25 Comparison of evaluation metrics on win10_VM (FindDLLInj Vs ProcInjectionsFind).

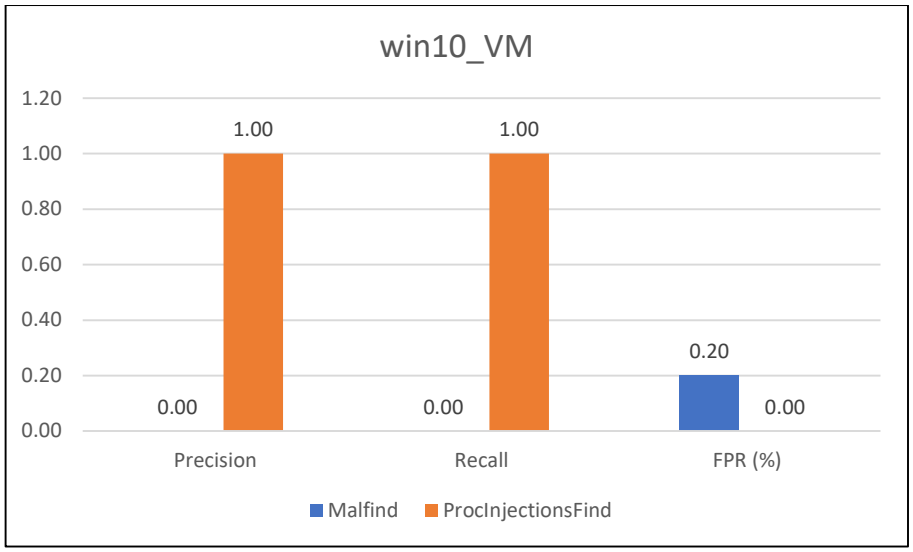


FIGURE 4.26 Comparison of evaluation metrics on win10_VM employing malware hiding technique (Malfind Vs ProcInjectionsFind).

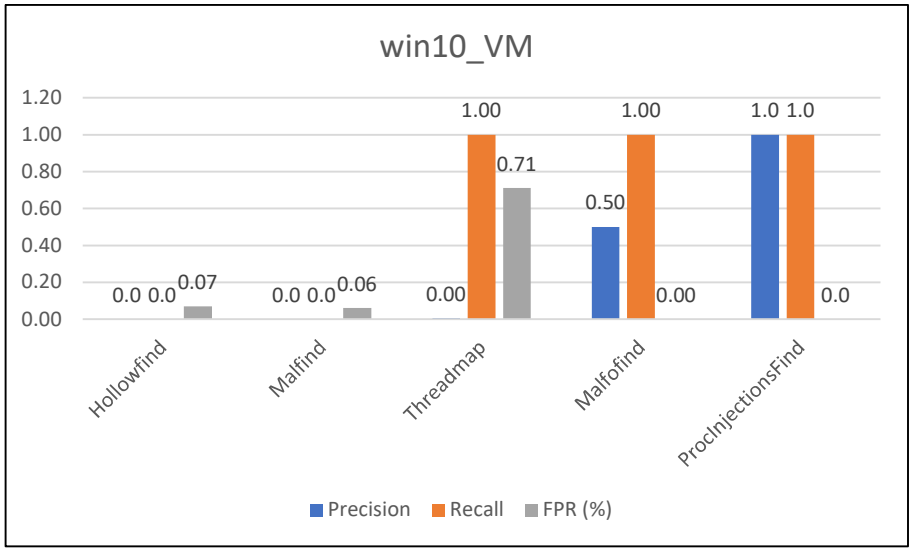


FIGURE 4.27 Comparison of evaluation metrics on win10_VM employing malware hiding technique (Hollowfind, Malfind, Threadmap, Malfofind Vs ProcInjectionsFind).

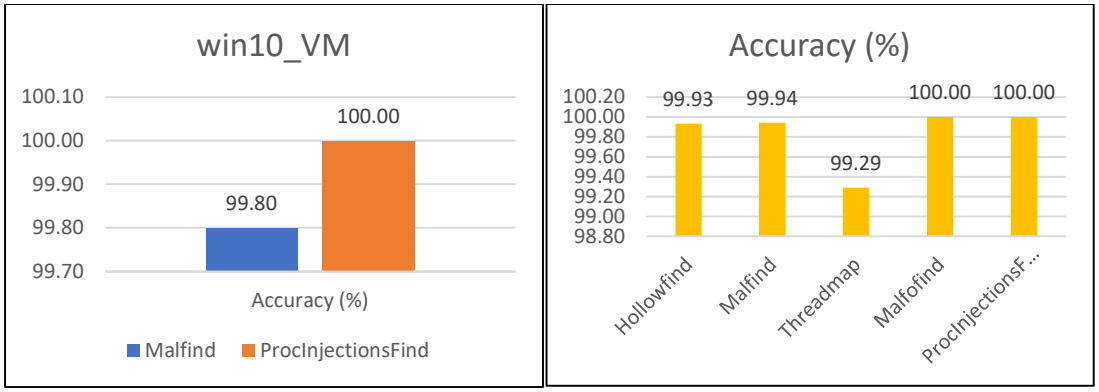


FIGURE 4.28 Comparison of accuracy on win10_VM employing malware hiding technique.

Figures from 4.7 to 4.28 shows a graphical representation of the derived results as depicted in Fig. 4.6. We utilize the ProcInjectionsFind plugin to locate an injected memory region and dump this injected memory region. VirusTotal is a powerful openly accessible free online malware scanner to get an analysis report of the examination that gives whether the tested executable file is malware or benign. It offers a free service for scanning suspicious files using several antivirus engines [105]. Hybrid Analysis is another free automated malware analysis service that detects and analyzes unknown threats using a unique Hybrid Analysis technology [106]. We may hash the memory region identified as an injected one and submit file hash to VirusTotal or Hybrid Analysis (online malware scanners) to verify our findings. Fig. 4.30 shows analyzing file hash of injected memory regions using Hybrid Analysis and Fig. 4.31 depicts analyzing file hash of injected memory regions using VirusTotal. Both these malware scanners, namely Hybrid Analysis and VirusTotal flagged this file as malicious.

```
dmt@dmt-HP-Laptop-15-da1xxx:~/Locate-Malicious-Process/vad_dump_info/cridex/malicious
dmt@dmt-HP-Laptop-15-da1xxx:~/Locate-Malicious-Process/vad_dump_info/cridex/malicious$ md5sum explorer.exe.23dea70.0x01460000-0x01480fff.dmp
16a6b5e927845866d8a57eb8b7cd718e  explorer.exe.23dea70.0x01460000-0x01480fff.dmp
dmt@dmt-HP-Laptop-15-da1xxx:~/Locate-Malicious-Process/vad_dump_info/cridex/malicious$
```

FIGURE 4.29 Calculating suspected file hash.

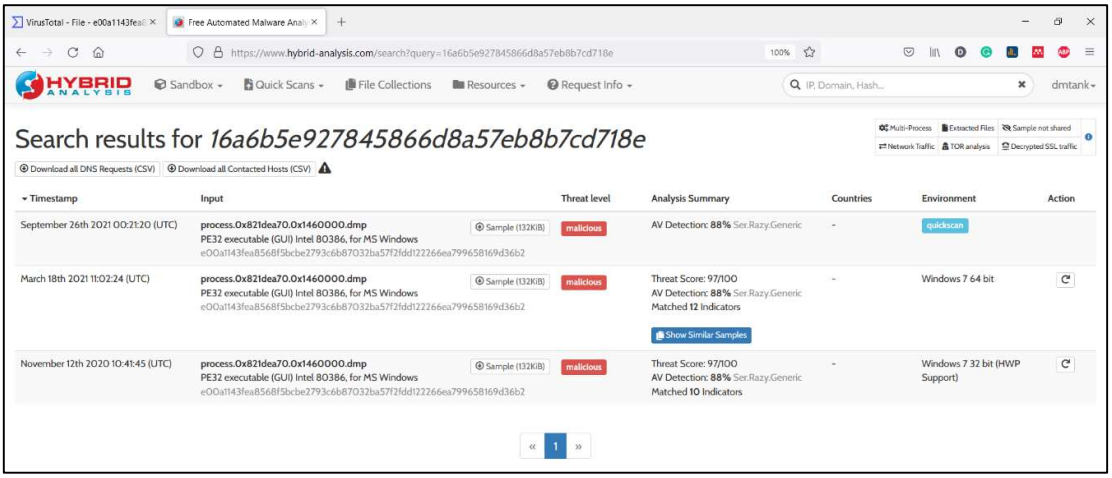


FIGURE 4.30 Analyzing suspicious memory region using Hybrid Analysis [106].

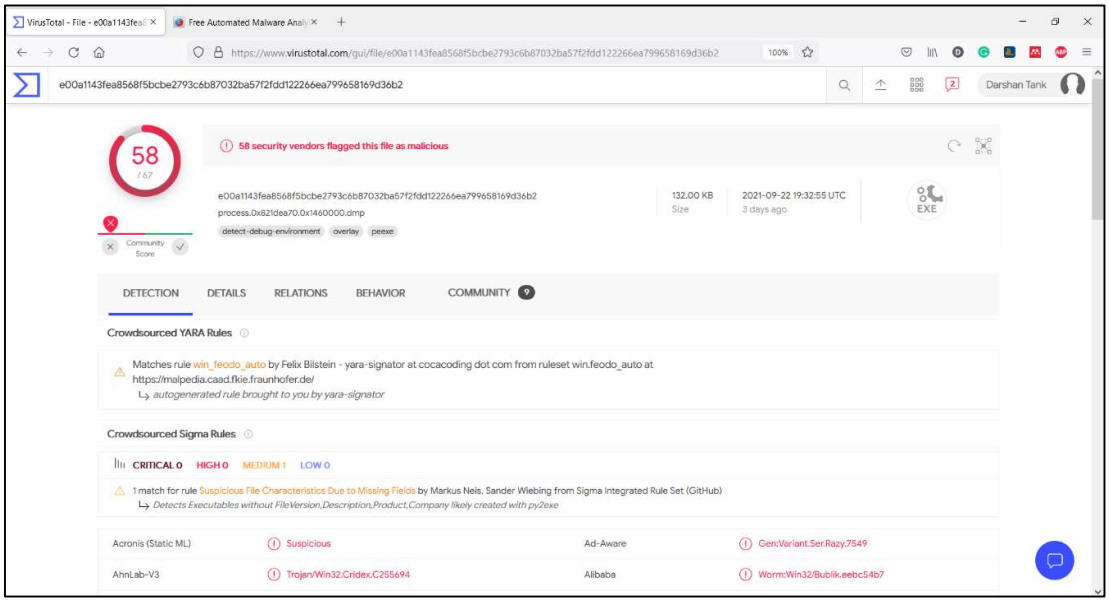


FIGURE 4.31 Analyzing suspicious memory region using VirusTotal [105].

```
dmt@dmt-HP-Laptop-15-da1xxx:~/Downloads/malwarecookbook/VirusTotal_API
dmt@dmt-HP-Laptop-15-da1xxx:~/Downloads/malwarecookbook/VirusTotal_API$ python vtlite.py -j 16a6b5e927845866d8a57eb8b7cd718e
```

FIGURE 4.32 Dumps the full VirusTotal report to file.

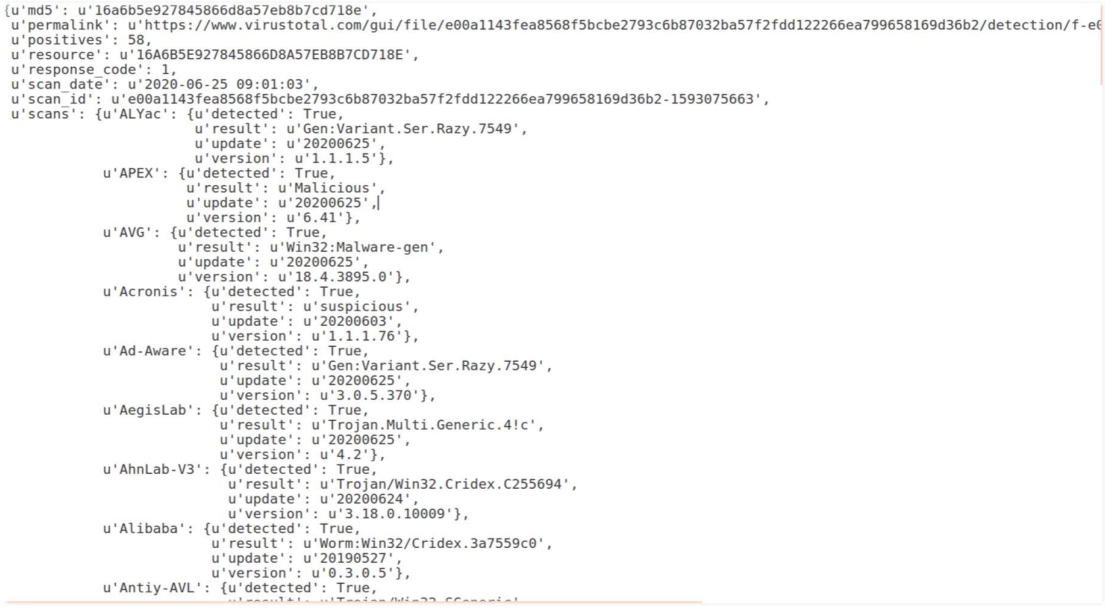


FIGURE 4.33 Preview of VirusTotal report.

We cross-checked our inferred results with VirusTotal. Fig. 4.32 shows the execution of the VirusTotal API script. This API script scans a file hash and consequently submits the file hash to VirusTotal (online malware scanner) and dumps the full VirusTotal report to the file. Fig. 4.33 shows the preview of this VirusTotal report. In this way, we don't need to upload malicious/injected memory sections online for the examination.

4.3.2 Containment Plan

In this section, we highlight a few countermeasures which can be adopted by security analysts, once the malware is identified inside the running VMs in the context of this work.

- Every KVM is just one separate Linux process.
- As one can't affect the live VM directly from the host, one can kill the KVM process from the host machine, i.e., "kill the virtual machine" → get_vm_pid_to "kill vm"
- Suspend or reset a malicious virtual machine.
- Stop a malicious virtual machine and save its configuration in order to restart it later.

4.4 Concluding Remarks

The experimental results show that the proposed approaches are promising. The inferred results reveal several insights. The proposed VMIPID model reports promising outcomes on each of the three Virtual Machines (Win7, Win8.1, & Win_10) in all assessment metrics. It reveals 100 % positive predictive value, completeness, f1-score, accuracy and 0% false positive rate on each of the three VMs. Experimental results demonstrate that the proposed

model achieves high accuracy and f1-score and has more true positives and fewer false positives when compared with other existing solutions. Overall, the proposed framework completely recognizes more malware families and stands over other frameworks in all assessment metrics characterized in this work. The ProcInjectionsFind module is designed to automate the detection of different process injection techniques described in this work and also dump recognized injected memory regions to disk for a detailed analysis and assessment.

CHAPTER – 5

5. Conclusion and Future Work

5.1 Conclusion

One of the primary contributions of this work is to accomplish live introspection of running VMs for possible indication of process injection. We adapt the Out-of-VM introspection approach thereby monitoring the runtime state of VMs completely outside of VMs, i.e., from the hypervisor. We implemented the proposed model (VMIPID) as a volatility plugin that can be independently called and runs against the windows memory image or memory of a live VM and analyzes every memory region to conclude if it is the consequence of process injection.

Our developed module certainly reports any malicious memory region containing injected code and we also evaluated it against the implementation of the hiding technique. Experimental outcomes show that our model classifies injected memory regions with high accuracy and completeness and has more true positives and fewer false positives when compared to other existing solutions.

Our proposed approach does not require signature-based analysis and in-depth knowledge of advanced memory forensics or operating system's internals. Our proposed framework neither modifies the guest OS at the host level nor impacts the performance of the host OS and VMs. The proposed system detects an actual injected memory region inside an injected process and thereby pinpointing the malware exactly. The proposed detection model recognizes injected memory regions into every infected process. The proposed VMIPID model reports promising results on each of the three Virtual Machines (Win7, Win8.1, & Win_10) in all assessment metrics and reduce false positive and false negative samples.

We utilized LibVMI python bindings integrated with the Volatility framework to analyze the memory of running VMs. Our proposed algorithm will search for injected memory region inside a victim/target process and consequently locate injected code or DLLs within the memory image. The ProcInjectionsFind module is intended to automate the detection of different process injection techniques described in this work. We can likewise dump recognized malicious memory regions with the ProcInjectionsFind plugin.

5.2 Scope of Future Work

Our study centers around the identification of eight unique executions of process injection. One could add more process injection techniques and cover other malware hiding techniques which are not covered in this study as future work. A developed plugin has been proved against Windows-based VMs, i.e., Windows 7, Windows 8.1 and Windows 10 respectively. The plugin may be proved against other windows-based operating systems to see its platform compatibility and conformity. Our volatility plugin (ProcInjectionsFind) can port over to another popular memory forensic framework known as Rekall [107] and could be integrated with existing process injection detection plugins. The proposed framework could likewise be ported to Linux or Mac OS based VMs.

For now, a DLL is correlated with one or more threads associating its load time with the thread creation time. This appears to work effectively, yet perhaps there is a more precise way to do the correlation that didn't reveal within this work. Future work concerns in-depth examination of different operating system's data structures and new perspectives to attempt different methods to locate injected memory regions that current methods/tools don't reveal. The proposed approach finds memory-only malicious code, i.e., injected code resident in memory. Future work requires further investigation to find injected code stored in the files on disk or scattered all over the paging files.

References

[1] Process Injection - Threat Detection Report - Red Canary, https://redcanary.com/threat-detection-report/techniques/process-injection/.

[2] G. Zhu, Y. Yin, R. Cai and K. Li, "Detecting Virtualization Specific Vulnerabilities in Cloud Computing Environment," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 743-748, doi: 10.1109/CLOUD.2017.105.

[3] Chandramouli, R. (2018), Security Recommendations for Hypervisor Deployment on Servers, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.SP.800-125A.

[4] ENISA, Threat Landscape Report 2017, ENISA, published 15 January 2018, ISBN 978-92-9204-250-9, ISSN 2363-3050, DOI 10.2824/967192, https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2017.

[5] ENISA, ETL (ENISA Thread Landscape) Web based tool, ENISA, https://etl.enisa.europa.eu/#/.

[6] Hal Pomeranz, Detecting Malware with Memory Forensics, SANS Webcast, Oct 2012, http://www.deer-run.com/~hal/, last accessed on 11 September 2018.

[7] D. O'Brien, "Internet Security Threat Report - Ransomware 2017," Symantec, July 2017.

[8] Guinet, "Wannakey: Wannacry in-memory key recovery," 2017. [Online]. Available: https://github.com/aguinet/wannakey.

[9] Symantec Security Response, "Can files locked by WannaCry be decrypted: A technical analysis," 2017. [Online]. Available: https://medium.com/threat-intel/wannacry-ransomwaredecryption-821c7e3f0a2b.

[10] White, A., 2013. Hashtest Volatility Plugin. https://github.com/a-white/Hashtest. Accessed 21 May 2018.

[11] H. Xiong, Z. Liu, W. Xu and S. Jiao, "Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM," 2012 IEEE 12th International Conference on Computer and Information Technology, 2012, pp. 549-556, doi: 10.1109/CIT.2012.119.

[12] Francia, G. A., Garrett, A., Brookshire, Jr., T. (2012). In H.R. Arabnia (Ed.), Virtualization for a Cyber-Security Laboratory. 2012 International Conference on Frontiers in Education: Computer Science and Computer Engineering.

[13] Paundu, A. W., Fall, D., Miyamoto, D., & Kadobayashi, Y. (2018). Leveraging KVM Events to Detect Cache-Based Side Channel Attacks in a Virtualization Environment. Security and Communication Networks, 2018, 1-18. DOI:10.1155/2018/4216240.

[14] Tank, D. M., Aggarwal, A., & Chaubey, N. K. (2020). Cyber Security Aspects of Virtualization in Cloud Computing Environments: Analyzing Virtualization-Specific Cyber Security Risks. In N. Chaubey, & B. Prajapati (Eds.), Quantum Cryptography and the Future of Cyber Security (pp. 283-299). Hershey, PA: IGI Global. doi:10.4018/978-1-7998-2253-0.ch013.

[15] Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., & Skalsky, N.C. (2010). HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. CCS '10.

[16] Wang, Z., & Jiang, X. (2010). HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. 2010 IEEE Symposium on Security and Privacy, 380-395.

[17] Zhang, F., Chen, J., Chen, H., & Zang, B. (2011). CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.

[18] Keller, E., Szefer, J., Rexford, J., & B. Lee., R. (2010). NoHype: virtualized cloud infrastructure without the virtualization. In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). Association for Computing Machinery, New York, NY, USA, 350–361. DOI:https://doi.org/10.1145/1815961.1816010

[19] Szefer, J., & Lee, R.B. (2012). Architectural support for hypervisor-secure virtualization. ASPLOS XVII.

[20] Xiong, H., Zheng, Q., Zhang, X., & Yao, D.D. (2013). CloudSafe: Securing data processing within vulnerable virtualization environments in the cloud. 2013 IEEE Conference on Communications and Network Security (CNS), 172-180.

[21] Lee, S., & Yu, F. (2014). Securing KVM-Based Cloud Systems via Virtualization Introspection. 2014 47th Hawaii International Conference on System Sciences, 5028-5037.

[22] Kumara M A, A., & Jaidhar, C.D. (2015). Hypervisor and virtual machine dependent Intrusion Detection and Prevention System for virtualized cloud environment. 2015 1st International Conference on Telematics and Future Generation Networks (TAFGEN), 28-33.

[23] Götzfried, J., Dorr, N., Palutke, R., & Müller, T. (2016). HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space. 2016 11th International Conference on Availability, Reliability and Security (ARES), 79-87.

[24] Tien, C., Liao, J., Chang, S., & Kuo, S. (2017). Memory forensics using virtual machine introspection for Malware analysis. 2017 IEEE Conference on Dependable and Secure Computing, 518-519.

[25] Tang, W., & Mi, Z. (2018). Secure and Efficient In-Hypervisor Memory Introspection Using Nested Virtualization. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 186-191.

[26] Zhang, S., Meng, X., Wang, L., Xu, L., & Han, X. (2018). Secure Virtualization Environment Based on Advanced Memory Introspection. Secur. Commun. Networks, 2018, 9410278:1-9410278:16.

[27] Yadav, Y., & Krishna, C. R. (2018). Two-Level Security Framework for Virtual Machine Migration in Cloud Computing. i-manager's Journal on Information Technology, 7(1), 34-44. https://doi.org/10.26634/jit.7.1.14095

[28] Li, S., Koh, J.S., & Nieh, J. (2019). Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. USENIX Security Symposium.

[29] Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu, "CBM: free, automatic malware analysis framework using API call sequences," in Knowledge Engineering and Management, pp. 225–236, Springer, Berlin, Germany, 2014.

[30] Li, C., Xiang, Y., & Shi, J. (2015). A Model of Dynamic Malware Analysis Based on VMI. ICA3PP.

[31] Ajay Kumara, M.A., & Jaidhar, C.D. (2015). Virtual machine introspection based spurious process detection in virtualized cloud computing environment. 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 309-315.

[32] Volatility Foundation: The Volatility Framework (2016). http://www.volatilityfoundation.org. Accessed 24 Apr 2017.

[33] Monnappa, K.A.: Detecting deceptive process hollowing techniques using hollowfind volatility plugin (2016). https://cysinfo.com/detecting-deceptivehollowing-techniques/. Accessed 25 Apr 2018.

[34] Pék, G., Lázár, Z., Várnagy, Z., Félegyházi, M., & Buttyán, L. (2016). Membrane: A Posteriori Detection of Malicious Code Loading by Memory Paging Analysis. ESORICS. LNCS, vol. 9878, pp. 199–216. doi:10.1007/978-3-319-45744-4 10.

[35] Hosseini, A. (2017, July 18). Ten process injection techniques: A technical survey of common and trending process injection techniques. Retrieved December 7, 2017.

[36] Code from "Taking Hunting to the Next Level: Hunting in Memory" presentation at SANS Threat Hunting Summit 2017 by Jared Atkinson and Joe Desimone - Github. Get-InjectedThread.ps1, https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2.

[37] Barabosch T., Bergmann N., Dombeck A., Padilla E. (2017) Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In: Polychronakis M., Meier M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2017. Lecture Notes in Computer Science, vol 10327. Springer, Cham. https://doi.org/10.1007/978-3-319-60876-1_10

[38] Block, F., & Dewald, A. (2019). Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries. Digit. Investig., 29-Supplement, S3-S12, ISSN 1742-2876, https://doi.org/10.1016/j.diin.2019.04.008.

[39] Mathew J., Ajay Kumara M.A. (2020) API Call Based Malware Detection Approach Using Recurrent Neural Network—LSTM. In: Abraham A., Cherukuri A., Melin P., Gandhi N. (eds) Intelligent Systems Design and Applications. ISDA 2018 2018. Advances in Intelligent Systems and Computing, vol 940. Springer, Cham. https://doi.org/10.1007/978-3-030-16657-1_9

[40] Rakotondravony, N., Taubmann, B., Mandarawi, W., Weishäupl, E., Xu, P., Kolosnjaji, B., ... & Reiser, H. P. (2017). Classifying malware attacks in IaaS cloud environments. Journal of Cloud Computing, 6(1), 26.

[41] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., & Lee, W. (2011, May). Virtuoso: Narrowing the semantic gap in virtual machine introspection. In 2011 IEEE symposium on security and privacy (pp. 297-312). IEEE.

[42] Fu, Yangchun & Lin, Zhiqiang. (2013). Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. ACM Transactions on Information and System Security. 16. 1-29. 10.1145/2516951.2505124.

[43] Process Injection, Technique T1055 - Enterprise | MITRE ATT&CK ® https://attack.mitre.org/techniques/T1055/, last accessed on 21 October 2019.

[44] Microsoft: Microsoft Malware Classification Challenge (BIG 2015) (2015). https://www.kaggle.com/c/malware-classification. Accessed 24 Apr 2017.

[45] Technopedia.com, Virtualization, https://www.techopedia.com/definition/719/virtualization, last accessed on 21 September 2019.

[46] Rajkumar Buyya, Christian Vecchiola, and S. T. Selvi. Mastering Cloud Computing. Morgan Kaufmann, 2013.

[47] H. Bengtsson and D. Hjerpe, Master Thesis on "Digital forensics - Performing virtual primary memory extraction in cloud environments using VMI", May 2018, www.diva-portal.org/smash/get/diva2:1230900/FULLTEXT01.pdf.

[48] Virtual Machine Introspection in Malware Analysis. https://resources.infosecinstitute.com/virtual-machine-introspection-in-malware-analysis/, Last accessed on 17 December 2029.

[49] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory (1st. ed.). Wiley Publishing.

[50] WindowsSCOPE, Cyber Forensics 3.2, http://www.windowsscope.com/windowsscope-cyber-forensics/, last accessed on 11 September 2018.

[51] F-Response, https://www.f-response.com/, last accessed on 11 September 2018.

[52] Introduction to LibVMI. http://libvmi.com/docs/gcode-intro.html, Last accessed on 11 January 2020.

[53] An advanced memory forensics framework. http://volatilityfoundation.org/, Last accessed on 17 November 2019.

[54] Finding Advanced Malware Using Volatility. https://eforensicsmag.com/finding-advanced-malware-using-volatility/, Last accessed on 11 January 2020.

[55] Memory Forensics Investigation using Volatility. https://www.hackingarticles.in/memory-forensics-investigation-using-volatility-part-1/, Last accessed on 11 January 2020.

[56] Bharati Ainapure, Deven Shah, and A. Ananda Rao. (2016). Performance Analysis of Virtual Machine Introspection Tools in Cloud Environment. In Proceedings of the In-ternational Conference on Informatics and Analytics (ICIA-16). Association for Com-puting Machinery, New York, NY, USA, Article 27, 1–6. DOI: https://doi.org/10.1145/2980258.2980309.

[57] Processes and Threads, https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads.

[58] What are the process handles in Windows? What are some examples of it? - Quora. https://www.quora.com/What-are-the-process-handles-in-Windows-What-are-some-examples-of-it.

[59] Modules - Windows driver | Microsoft Docs, https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/modules.

[60] Geoff Chappel, LDR_DATA_TABLE_ENTRY, https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntldr/ldr_data_table_entry.htm.

[61] What is Kernel32.dll? - Definition from Techopedia, https://www.techopedia.com/definition/3379/kernel32dll.

[62] Microsoft, LoadLibrary and AfxLoadLibrary, https://msdn.microsoft.com/en-us/library/zzk20sxw.aspx.

[63] Hooking Series PART I: Import Address Table Hooking - ReLearEx, https://relearex.wordpress.com/2017/12/26/hooking-series-part-i-import-address-table-hooking/, Posted on Dec 26, 2017.

[64] B. Dolan-Gavitt, "The VAD tree: A process-eye view of physical memory," Digital Investigation, vol. 4, pp. 62–64, 2007.

[65] Wikipedia, Portable Executable, https://en.wikipedia.org/wiki/Portable_Executable.

[66] Matt Pietrek, Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, written on March 1994, https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[67] Microsoft, PE Format, https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format, uploaded on 31 March 2021.

[68] WIKIPEDIA, DLL injection, https://en.wikipedia.org/wiki/DLL_injection, edited on 18 September 2018, last accessed on 1 October 2018.

[69] Michael Sikorski and Andrew Honig, PRACTICAL MALWARE ANALYSIS, no starch press, 2012, ISBN-10: 1-59327-290-1, ISBN-13: 978-1-59327-290-6.

[70] Balaoura S. (2018) Process Injection Techniques and Detection using the Volatility Framework. Master's Thesis. University of Piraeus.

[71] Rio Asmara Suryadi, September 6, 2020, Basic Remote Thread Injection - Cyber Security | Penetration Test | Malware Analysis, https://rioasmara.com/2020/09/06/basic-remote-thread-injection/.

[72] Jared Atkinson and Joe Desimone, Taking Hunting to the Next Level - Hunting in Memory presentation, Endgame, SANS Institute Threat Hunting and IR Summit (April 2017), https://www.sans.org/summit-archives/file/summit-archive-1492714038.pdf, last accessed on 1 October 2018.

[73] Luis Rocha, Malware Analysis – Dridex & Process Hollowing, https://countuponsecurity.com/2015/12/07/malware-analysis-dridex-process-hollowing/, uploaded 07 December 2015, last accessed on 1 October 2018.

[74] Microsoft. (n.d.). Asynchronous Procedure Calls. Retrieved December 8, 2017.

[75] Process Injection: APC Injection - Malware - 0x00sec - The Home of the Hacker, https://0x00sec.org/t/process-injection-apc-injection/24608.

[76] Cyber security info notes — ENISA, https://www.enisa.europa.eu/publications/info-notes/atombombing-2013-a-new-code-injection-attack.

[77] AVTEST. The AV-TEST Security Report 2016/17. Tech. Rep.; 2017. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf.

[78] Garnaeva M, Sinitsyn F, Namestnikov Y, Makrushin D, Liskin A. Overall statistics for 2016; https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf.

[79] Symantec. Internet security threat report 21. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[80] Volatility, [online] Available: http://www.volatilityfoundation.org/.

[81] KVM, [online] Available: https://www.linux-kvm.org/page/Main_Page/.

[82] LibVMI, [online] Available: http://libvmi.com/.

[83] VirtualProtectEx function (memoryapi.h) - Win32 apps. Microsoft Docs (2018, December 5). https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotectex.

[84] K. Kourai and K. Nakamura, "Efficient VM Introspection in KVM and Performance Comparison with Xen," 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing, 2014, pp. 192-202, doi: 10.1109/PRDC.2014.33.

[85] Openstack. https://www.openstack.org/. Accessed 02 May 2018.

[86] D. M. W. Powers, “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation,” Mach. Learn. Technol., vol. 2, no. 3, pp. 121–130, 2008.

[87] Y. Sasaki, “The truth of the F-measure,” Teach Tutor mater, vol. 1, no. 5, pp. 1–5, 2007.

[88] False Positive Rate | Split Glossary, https://www.split.io/glossary/false-positive-rate/.

[89] The Volatility Foundation, 2017. Volatility's Malfind Plugin. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py. Accessed 24 Apr 2018.

[90] Monnappa, K.A., 2016. HollowFind Volatility Plugin. https://github.com/monnappa22/HollowFind/blob/master/hollowfind.py. Accessed 24 Apr 2018.

[91] KSL group, 2017. Threadmap Volatility Plugin. https://github.com/kslgroup/threadmap. Accessed 25 Apr 2018.

[92] Pshoul, D., 2017. Malfofind Volatility Plugin. https://github.com/volatilityfoundation/community/blob/master/DimaPshoul/malfofind.py. Accessed 25 Apr 2018.

[93] Memory Dump on Windows 7 64-bit, KSLSample.vmem, Process hollowing in different approaches, Available online at https://www.mediafire.com/file/jlmtbbinanuh6jr/KSLSample.rar.

[94] The Volatility Foundation, 2017. Volatility's Vadinfo Plugin. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/vadinfo.py. Accessed 26 Apr 2018.

[95] The Volatility Foundation, 2017. Volatility's Impscan Plugin. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/impscan.py. Accessed 26 Apr 2018.

[96] The Volatility Foundation, 2017. Volatility's Volshell Plugin. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/volshell.py. Accessed 26 Apr 2018.

[97] Injection techniques. GitHub – theevilbit/injection. https://github.com/theevilbit/injection.

[98] Reflective DLL Injection. GitHub – stephenfewer/ReflectiveDLLInjection. https://github.com/stephenfewer/ReflectiveDLLInjection.

[99] Process Hollowing. GitHub – m0n0ph1/Process-Hollowing. https://github.com/m0n0ph1/Process-Hollowing.

[100] Seven different DLL injection techniques. GitHub – fdiskyou/injectAllTheThings. https://github.com/fdiskyou/injectAllTheThings.

[101] Process Injection Techniques. GitHub – secrary/InjectProc. https://github.com/secrary/InjectProc.

[102] AtomBombing. GitHub - BreakingMalwareResearch/atom-bombing: Brand New Code Injection for Windows. https://github.com/BreakingMalwareResearch/atom-bombing.

[103] Windows Process Injection. GitHub - CptGibbon/Windows-Process-Injection: Some simple process injection techniques targeting the Windows platform. https://github.com/CptGibbon/Windows-Process-Injection.

[104] Code Injection Tools. DFRWS-USA-2019/tools at master · f-block/DFRWS-USA-2019 · GitHub - https://github.com/f-block/DFRWS-USA-2019/tree/master/tools.

[105] VirusTotal, [online] Available: https://www.virustotal.com/.

[106] Hybrid Analysis, [online] Available: https://www.hybrid-analysis.com/.

[107] Google Inc, 2018. Rekall memory forensic framework. http://www.rekall-forensic.com. Accessed 25 Sept 2019.

List of Publications

1. Tank D., Aggarwal A., Chaubey N. (2020). A Method for Malware Detection in Virtualization Environment. In: Chaubey N., Parikh S., Amin K. (eds) Computing Science, Communication and Security. COMS2 2020. Communications in Computer and Information Science, vol 1235, Published by Springer, Singapore, ISBN: 978-981-15-6647-9, https://doi.org/10.1007/978-981-15-6648-6_21

2. Tank, D. M., Aggarwal, A., & Chaubey, N. K. (2020). Cyber Security Aspects of Virtualization in Cloud Computing Environments: Analyzing Virtualization-Specific Cyber Security Risks. In N. Chaubey, & B. Prajapati (Eds.), Quantum Cryptography and the Future of Cyber Security (pp. 283-299). Hershey, PA: IGI Global. doi:10.4018/978-1-7998-2253-0.ch013

3. Tank, D., Aggarwal, A., & Chaubey, N. (2019). Cache attack detection in virtualized environments. Journal of Information and Optimization Sciences, 40:5, 1109-1119, DOI: 10.1080/02522667.2019.1638001, ISSN: 0252-2667, Abstracted and Indexed in ESCI® (Web of Science)

4. Tank, D., Aggarwal, A. & Chaubey, N. (2019). Virtualization vulnerabilities, security issues, and solutions: a critical study and comparison. International Journal of Information Technology. Published by Springer, ISSN: 2511-2104, https://doi.org/10.1007/s41870-019-00294-x, Abstracted and Indexed in UGC-CARE List (India)

5. Tank, D. M. (2017). Security and Privacy Issues, Solutions, and Tools for MCC. In K. Munir (Ed.), Security Management in Mobile Cloud Computing (pp. 121-147). Hershey, PA: IGI Global. doi:10.4018/978-1-5225-0602-7.ch006

6. Tank, D., Aggarwal, A., & Chaubey, N. (2017). Security Analysis of OpenStack Keystone. International Journal of Latest Technology in Engineering, Management & Applied Science (IJLTEMAS), Volume 6, Issue 6; pp. 31-38.

7. Chaubey, N. K., Tank. D. M. (2016). Security, Privacy and Challenges in Mobile Cloud Computing (MCC): - A Critical Study and Comparison. International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE), Volume 4, Issue 2. DOI: 10.15680/IJIRCCE.2016.0402028; pp. 1259-1266.