Behaviour Anomaly on Linux Systems to Detect Zero-day Malware Attacks

Ovais Ahmed

A thesis submitted to the Faculty of Design and Creative Technologies Auckland University of Technology

In partial fulfilment of the requirements for the degree of Master of Information Security and Digital Forensics

School of Engineering, Computer and Mathematical Sciences
Auckland, New Zealand, 25 February 2022

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a University or other institution of higher learning except where due acknowledgement is made in the acknowledgements.

Ovais Ahmed

29-11-2021

Abstract

Internet-connected devices have been the subject of cyber threats due to the gain malicious actors can get by compromising these systems. Endpoint protection is available on these systems, protecting if the malware signature is available for the malicious software. The challenge is that if the signature is not available on the endpoint protection, as in the case of zero-day malware, the endpoint will not detect or protect the system. The system follows the file analysis of zero-day malware in the sandbox environment for file identification, creating the signature and updating the endpoint database. The process of zero-day can generate a delay which can result in substantial damage to the systems by the time signature is updated. The research examines the abnormal behaviour on a Linux-based operating system and evaluates the method to explore the zero-day malware build for the platform.

Malware samples are sourced from the available public repositories. The sample files used include known malicious and known non-malicious files. The known malicious files have the signatures available on the antivirus tool. Therefore, the setup removes the necessary signatures for the known malware sample files to treat them as zero-day malware. Total twenty-two malware has been used to test the method to detect the zero-day malware, out of which few have been tried without signature information on endpoint antivirus to determine the consistency of the test results.

The research examines the malware behaviour on the Linux based system. It monitors the process in the two different situations where non-malicious and known malware is executed at different intervals. The abnormal process behaviour detects the malicious file. The second phase of the research explores the methods to act on the file after the detection. It discusses YARA rules and programable interface integration across the platform to automate the file quarantine feature.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Background and Motivation

Malware is a computer code that is designed to disrupt, disable, or take control of computer systems. It is usually hidden in a regular file disguised as a harmless application (Christodorescu et al., 2007). It takes advantage of technical flaws and vulnerabilities in the hardware, operating systems, and software. The spread of such malicious software has become very common with easy access to the Internet, especially when business services are moving towards the public cloud (Joseph & Mukesh, 2019). As the business services head to an online platform, malicious actor canvas is increasing very rapidly. The threat vector has risen sharply. The online payment systems, including online credit card usage, phishing attacks for credential leaks, ransomware, and tools like key loggers that can be downloaded while browsing compromised websites, are a few types of attack vectors. The protection of such attacks is based not only on the security controls but also on end-user awareness. Many users unknowingly click a malicious link or browse a malicious site that may download malware in the background without user knowledge. The malicious code can execute in the background without the end-user noticing the behaviour. The processes are executed behind the legitimate application. Therefore end-point protection is desirable to prevent users from being victims of malicious activity.

Windows and Linux are two platforms that are used for client-server connectivity. In previous days popular platforms, due to their more significant presence, were more vulnerable to malware attacks, such as the end-point market is pretty much dominated by Microsoft based operating systems. Now Linux systems malware are getting popular as Linux operating systems are becoming more familiar with technology like IoT. The world is moving towards an intelligent world. More and more IoT devices will become

part of our daily use. It will increase the risk of the malicious actor taking advantage of unprotected systems. The Windows operating system is always considered more vulnerable to malware than Linux systems, and engineers do not tend to run the OS patching like windows considering it is more secure (Wu et al., 2012). It is not valid anymore as there is sample malware like PHP backdoor, rootkit and many others available for Linux based systems due to its widespread usage in IoT systems (Dmitry & Elena, 2020). Linux servers are also becoming part of the more extensive ecosystem where connected end-points are comprised of Linux and Microsoft operating systems. The Linux machine can store the malicious code, which can be exploited on the Windows machine. The attack techniques like server-side attack forgery where the request is sent to one machine to compromise the connected machines; hence, an ecosystem where all systems are interconnected requires serious protection from malware. Though the Linux system has more security control than the Windows system, like file permission directory permission, attackers can still bypass these controls to execute malware (Yaswinski et al., 2019). Security practices like Linux server patch management, regular security audits and logs monitoring, along with security hardening (Yaswinski et al., 2019), even cannot detect the malicious code. Antivirus software installed on Linux systems helps detect and remove the malware for files with signatures available for those malicious files (Mohanta et al., 2020). The way traditional antivirus system works is based on the signature updates, which gets the feeds from multiple threat intelligence (Martin et al., 2018). Recent ransomware attacks are an excellent example of compromising an end-point where the executable file encrypts the hard disk of the user machine. If such malware does not have known signature patterns, then the end-point protection will not be able to detect or remove it from the machine. The challenge is that if a new malware or malicious code or zero-day malware code is sitting on the system, which is not executed yet, traditional antivirus tools will not detect due to signature unavailability (Shah & Singh, 2016).

The lack of detecting zero-day malware poses a considerable threat for a single end-point and the entire ecosystem for connected end-points.

Hence the objective is to find a method to detect the execution of the malicious code and find the abnormal behaviour to protect the system. Documentation is available for traditional signature-based detection for the malicious code. The methods and techniques are available to handle the variant of the existing malicious file; however, considerable work is required to detect zero-day malware. This research thesis aims to analyse the gap in the detection of known and zero-day malware. The malicious file can cause a threat when it is executed. The point when the file is executed can be addressed to detect the zero-day malware. The research on the understanding of the malicious file behaviour is kept as the objective of the thesis. The motivation is to understand the behaviour and apply the control mechanism to limit the activity by quarantining the malicious file. The work focuses on methods and processes using multiple tools to identify the system processes initiated by the malicious code. In order to support the thesis work, a lab setup is prepared to experiment with the known malware sources and simulated that malware as zero-day malware. The malware file execution monitors the behaviour of the files and compares the activities with the expected behaviour. The thesis does not include the machine learning techniques for abnormal behaviours or the detection of false positives but focuses on the technique to detect zero-day malware due to abnormal behaviour. Open-source tools are used during the research for the Linux operating systems. Tools are readily available and customisable as per the experiment requirements. The environment is kept very controlled to avoid the spread of malware when executed. The toolsets used are customizable as per the lab environment. The selection of database and programming language to build the setup is only to support the method and technique discussed in chapter 3.

The research questions are raised to determine if the processes behaviour can detect the zero-day malware, and the second phase answers the automated method to act on the malicious file. The focus of the research is on abnormal behaviour of the processes

on the Linux based operating systems. The technique discussed in chapter 3 applies to the Windows based systems as well. However, the architecture of the Windows operating system needs to be considered while applying the same technique, which is not part of the thesis.

1.2. Thesis Structure

The thesis is divided into six parts. The first chapter has given the introduction of the thesis topic and discusses its background details. The motivation for conducting this research was briefed, and the reason for this research is emphasised. It gives a brief overview of the threat posed by zero-day malware on the system. Also, set the scope of the thesis work.

Chapter 2 presents the literature review and details the concepts involved with the Linux operating system internal architecture. The internal architecture is a critical component to study as the behaviour of the file is detected through processor utilisation. It provides the details for the malware and its analysis techniques to prevent the end-point with the malicious code. YARA rule concept is discussed. Challenges are identified in the literature, which elaborates the problems with the malware, which is zero-day.

Chapter 3 details the research questions, which are to be answered by the end of the thesis. The chapter details the method and technique used to build the lab setup to provide answers to the research question.

Chapter 4 provides the details of the test results-driven for the environment built in chapter 3. It includes step by step output for the method and technique defined in the previous chapter. Results of some of the sample malware have been presented in chapter 4 and provide an overview of the sample malware used for the experiment.

The discussion around the test results is presented in chapter 5. It also highlights the relation between the challenges highlighted in the literature view and the research question raised for this thesis.

The summary of the finding, limitations of the thesis and possible future areas are discussed in chapter 6.

2. Literature Review

This chapter discusses the components which are involved during the research. The first section provides the details for the existing malware analysis techniques and the variations. It also discusses the available mechanism for the protection on the endpoint with antivirus tools. It covers current tools to protect the malware and how multiple threat intelligence collaborates to update signatures for new malicious code. Different methods used by multiple vendors to update the database is discussed. The primary research focuses on zero-day malware detection on Linux operating system; therefore, this section contains a brief description of the internal architecture for the Linux operating system. Linux is an open-source platform with multiple vendor-specific variations; however, the base internal kernel and architecture is the same. The chapter ends with an overview of the related work for the detection of zero-day malware protection.

2.1. Malware Analysis

Malware breaches cost companies of all sizes. IBM security cost of Data Breach Research Report shows that the average accumulated cost is $3.86 million and alone in the United States is $8.64 million (Klaus & Elzweig, 2020).

Malware is malicious code designed to install covertly and target systems as undetected. It can be designed to destroy the data or install additional programs or exfiltrate it (Or-Meir et al., 2019). It may include all the above three depending on the target agenda, like espionage, cyber terrorism, or others. It is to compromise the confidentiality, integrity and availability of a victim's data. Malware has been evolving, and historically these malicious codes were designed to take immediate action and were easy to notice. Previously, such malware was easy to categorise depending on the various type of infection to follow the handling procedure. Such malware is designed for stealth and stays in the system silently or dormant for weeks or months; hence tough to be noticed and spread slowly over time

(Radhakrishnan et al., 2019). This way, it gets more time to gather more information to exfiltration. These attack types usually need one procedure, and most attacks are blended and use multiple methods. There are multiple transmission methods, including physical access to the system, social engineering, phishing, or visiting a malicious site.

Also, there are multiple forms of malware like viruses, worms, Trojan horses, mobile attacks and blended attacks. Viruses are self-replicating and install themselves into compiled viruses and interpreted viruses. Compiled viruses are executed by the OS and can be an infected file or boot sector virus. This category can trigger during the system boot and hence can cause harm even before starting the antivirus program (Mohanta et al., 2020). An application like macro and scripts executes the interpreted viruses. Worms are also self-replicating programs like viruses and are usually self-contained, which can execute and spread without user interaction (Christodorescu et al., 2007). There are two main categories of worms, network service worms and mass-mailing worms. Network service worms exploit network vulnerability to propagate and infect others. Mass mailing worms only particularly to email systems to spread and target others. Trojan horse programs are different from viruses or malware as they cannot be replicated. These trojan horses have hidden payloads. Such malware is usually disguised in a program and application which seems to be legit. Such software replaces legitimate files with malicious files or adds additional components to track the host, like keyloggers. The last attack vector to consider is the malicious mobile code. It is delivered remotely and executed on a local host without its intervention. Java, ActiveX and VB scripts languages are primarily used for such types of attacks. Blended attacks use a combination of viruses, worms or trojan horses (Gandotra et al., 2017).

There are different types of malware analysis; static analysis is analysing malware without executing or running it. The purpose of static analysis is to extract as much metadata from the malware as possible. In the dynamic analysis, malware is executed to understand the functionality and its behaviour. It is also performed in a containerised way to avoid exploitation. Code analysis is the process of reviewing the code. The review

of code can be accompanied either using the static or dynamic method. Behavioural analysis monitors the process execution, registries update and look for abnormal network behaviours (Or-Meir et al., 2019).

The goal of malware analysis is to prepare with the available toolsets, and these tools can help identify the malware. The next step after identification is to contain the malware and limit its impact. The mitigation is performed to reduce the risk and recover the services affected by the malware. It is always advisable to document the analysis and implement a plan to control such malware to exploit the vulnerability. The open-source platforms are more vulnerable to malware due to being publicly available. The malware defence on the system is protected by endpoint protection which involves anti-virus tools. Due to the reliance on signature-based alerts, the analyst will need to manually analyse and update the signature after investigation. There are several malware analysis tools available, like Cuckoo and Joe Sandbox.

2.2. End-Point Protection

Antivirus software provides end-points protection, which connects to the centralized repositories and downloads the signatures for the known malware. The software programs scan the end-point and match the files against those signatures. If files are found with the matching condition, the files are quarantined or removed from the system (Mohanta et al., 2020). The file removal process is also critical, as in the case that the file is already executed or engaged in some other processes, the file will not delete (Mcafee, 2021). However, such software tools are capable of alerting if the malicious files cannot be deleted. The limitation of these end-point protection methods is the dependency on malware's signature (Gandotra et al., 2017). If the vendor-specific does not have the updated database for signature, the client will not have details for the malicious file. If the malicious file with no signature update executes codes, then its impact will be zero-day malware. The zero-day malware will not be blocked by traditional

antivirus software (Crowdstrike, 2021). The attacker uses multiple dynamic techniques, which are challenging to block if the antivirus tool does not have available information. It leaves a security gap for the end-point protection if the malware is zero-day.

2.3. Cyber Threat intelligence

It is gathering, evaluating, and analysing data on threats that are being faced. It then allows strategizing the defence, which can assist in entirely preventing the threat or reducing the impact of the damage caused. Finally, understand the full details of the threat, including the tools used during the attack, what information is compromised, the method used for the malware communication, and others. This information is used to create a threat profile that can assist in the model of the impact of a specific attack that can be used to prevent and reduce the impact of attacks (Baker, 2019). It enables to get the multiple feeds from the public, private, dark web and aggregate the data on the threat ATP which impacts the business. There is malware information sharing platform available like virustotal, MISP and others. Such platforms exchange and share threat intelligence, an indicator of compromise (IoCs) about the targeted malware and attacks, or any intelligence within the community (Baker, 2019). Threat intelligence is the collection of databases maintained by the different security vendors. It ensures that if the one vendor marks the file malicious, it generates the hash, MD5 or SHA, and shares the information with the partners. It also validates the authenticity of the malicious file. As more vendor marks the file malicious, the more the confirmation of its malicious nature is verified. Threat intelligence is a process that transforms the file collected for the analyses till the final intelligence decision. It provides a guide to the cybersecurity team (Crowdstrike, 2021).

2.4. Linux operating System

This section covers the architecture of a Linux based system. It will include the dependencies of the operating system with the hardware, services, API and system process. These areas are essential to understand as vulnerabilities in these components can be utilized to exploit the operating system. Assessing the abnormal behaviour of the files that have dependencies on the system's architecture is critical to understand. Memory, CPU usage, and file location are the essential parts of detecting abnormal behaviour. These components can be monitored to contain zero-day malware.

2.4.1 System Architecture

As shown in figure 2.1., Linux operating system has three major components. Kernel, kernel modules and system libraries. The kernel is the core of the operating system, and it enables the communication between the device and the software, which is also responsible for managing the system resources. It includes four significant responsibilities device management, memory management, manages processor for system handling call. The Device management involves managing any Input-Output (I/O) device where devices include Graphics cards, Sound cards, and others. The kernel stores and use device drivers to manage the hardware. It also enables communication between different sets of hardware. Memory and process management has been described in more detail in section 2.5.2.1 and 2.5.2.2, respectively. Any query made by a programmer to the kernel is handled by the function called system handling call. The programmer needs system libraries to pass the message to the kernel. There are variations of system libraries available for different kernels.
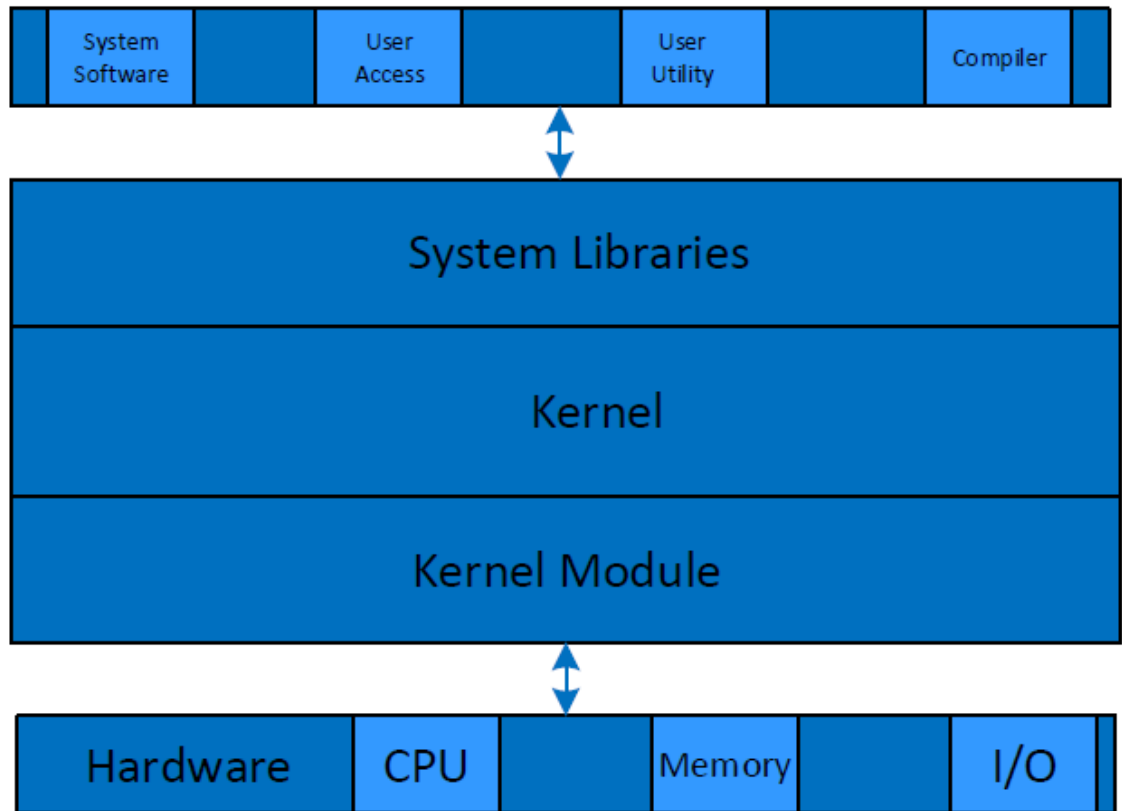
Figure 2. 1. *Linux System Architecture.*

(Ko et al., 2008)

Kernel modules manage the hardware layer, which includes memory, CPU and peripheral devices. The kernel module manages physical memory using a memory management unit. It uses virtual memory construct to use memory address space which is more than the physical memory. It is responsible for mapping virtual memory to physical memory, memory cache, and during the context switching, it manages the table of process pages.

Kernel module manages CPU resources by scheduling access to resources from processes. It is performed by enforcing the scheduling method to access the CPU while keeping the programs transparent for the CPU resource usage. As part of the algorithm, it creates, executes, suspends, and destroys the process. This component also works with the memory management to give CPU resource access for active requests in the memories.

2.4.2.1 Memory Management

2.4.2.1.1 Virtual Memory

Physical memory has a limited resource, and the hardware limits the memory size that can be installed on the system. The address range for the physical memory may not be contiguous, and these ranges vary with different system architecture (Stazi et al., 2017). Virtual memory is used to overcome the complexity. It provides large address spaces, which means that the system appears to have more memory than the actual available address. It enables the protection as it makes each process have dedicated virtual memory address space (Kim et al., 2014). Figure 2.2 depicts the impact of the system without virtual memory and how virtual memory overcomes the problem.
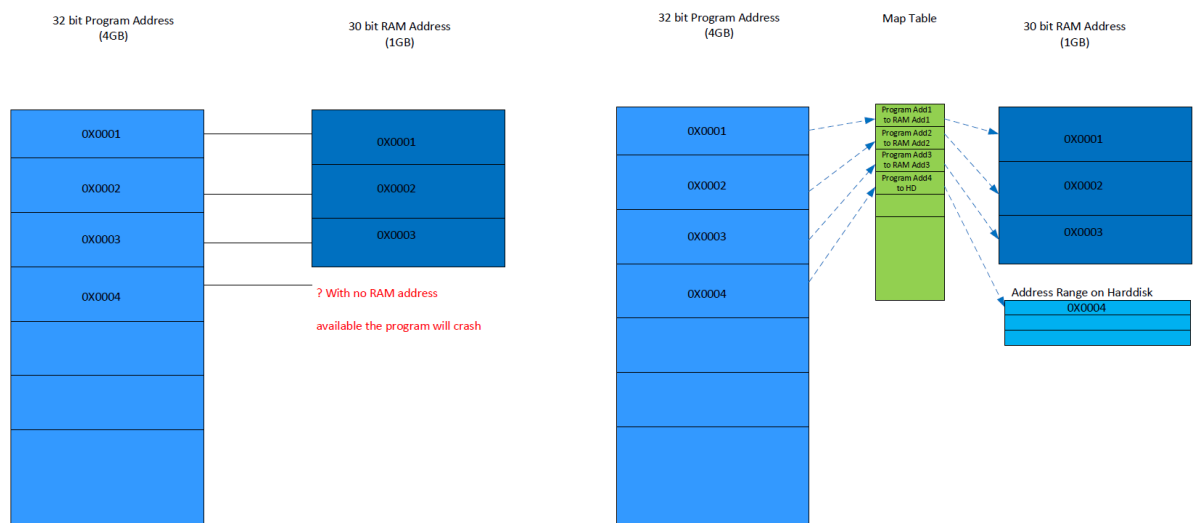
Figure 2. 2. Memory Address Table

(Ko et al., 2008)

The process allocated for each application is separate due to the complete virtual isolation of address space from one another. This segregation protects memory areas if the rogue application tries to overwrite the data on another legitimate application (Kim et al., 2014). It manages virtual memory for shared processes as well. This situation keeps

the shared process in the physical memory and logically shares the virtual address space. It does the memory mapping in which the file's content is linked to the address space of the virtual memory used by the process. Virtual memory also plays an essential role if the memory address is not contiguous. The map table can adjust the non-continuous address space by splitting the data content (Ko et al., 2008). The program will still see it as a single address. The figure 2.3 depicts the map table.
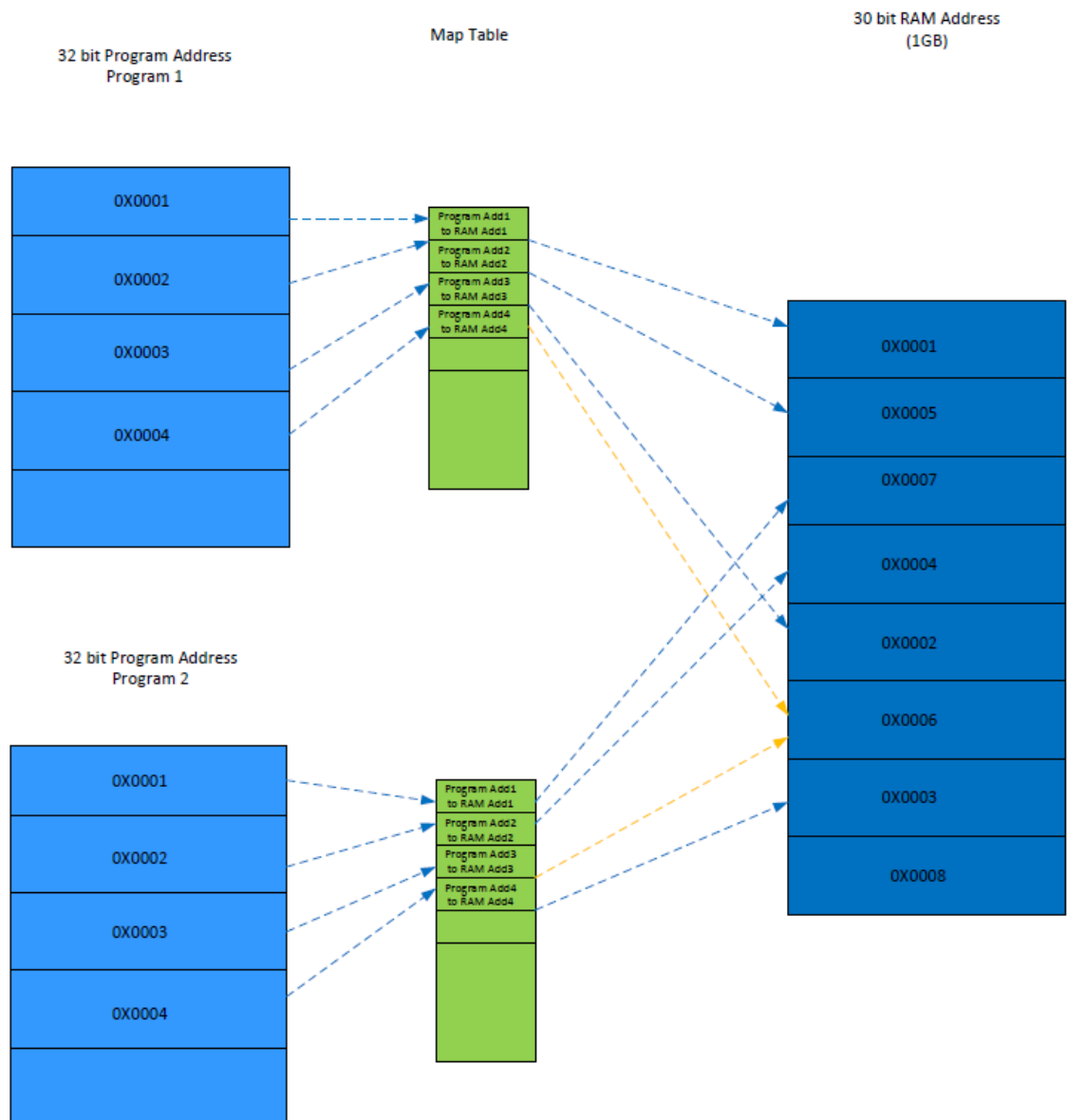
Figure 2. 3. *Map Table*

(Kim et al., 2014)

2.4.2.1.2 Abstract model of virtual memory

When the processor executes the program, it takes the instruction from memory and then decodes it. The data is fetched or stored from or in the memory during decoding (Ewais et al., 2016). The memory location is accessed via a virtual address which is mapped with the physical address. The processor is responsible for converting the virtual address to a physical address using the table maintained by the Linux operating system. The mapping is divided into pages to make it easier to maintain virtual and physical memory. The pages' size may vary depending on the operating systems and preferred the same page size to reduce the complexity. On the intel x86 system, the page size is 4K bytes. The page frame number is used to assign a unique number to each page (PFN). On the intel system, the virtual address contains two components, bits 0 to 11 (12 bits = 4096B (4KB)) makes the offset and the remaining bits represent the virtual page frame numbers. Table 2.1 shows a page table with a page of 4KB. It shows that if the program accesses the address 20, then it is in the address range of 0-4069, which set the offset of 0+20; hence on the virtual table, it maps as 20. Similarly, it adds 20 offsets for the corresponding physical address. In this case, the physical address will be 4100, as shown in figure 2.4.

Table 2. 1 Virtual Address MAP

Fine Grain - MAP each address		Coarse Grain - MAP Chunk of 4KB Pages	
MAP		MAP	
Virtual Address	Physical Address	Virtual Address	Physical Address
0X00001	0X00022	0-4095	4096-8191
0x000AB	0X00021		
0X0001C	0X00001		

12287

4KB

8192

4 bytes offset same for physical and
virtual address

Physical Address Space

4095

4KB

0
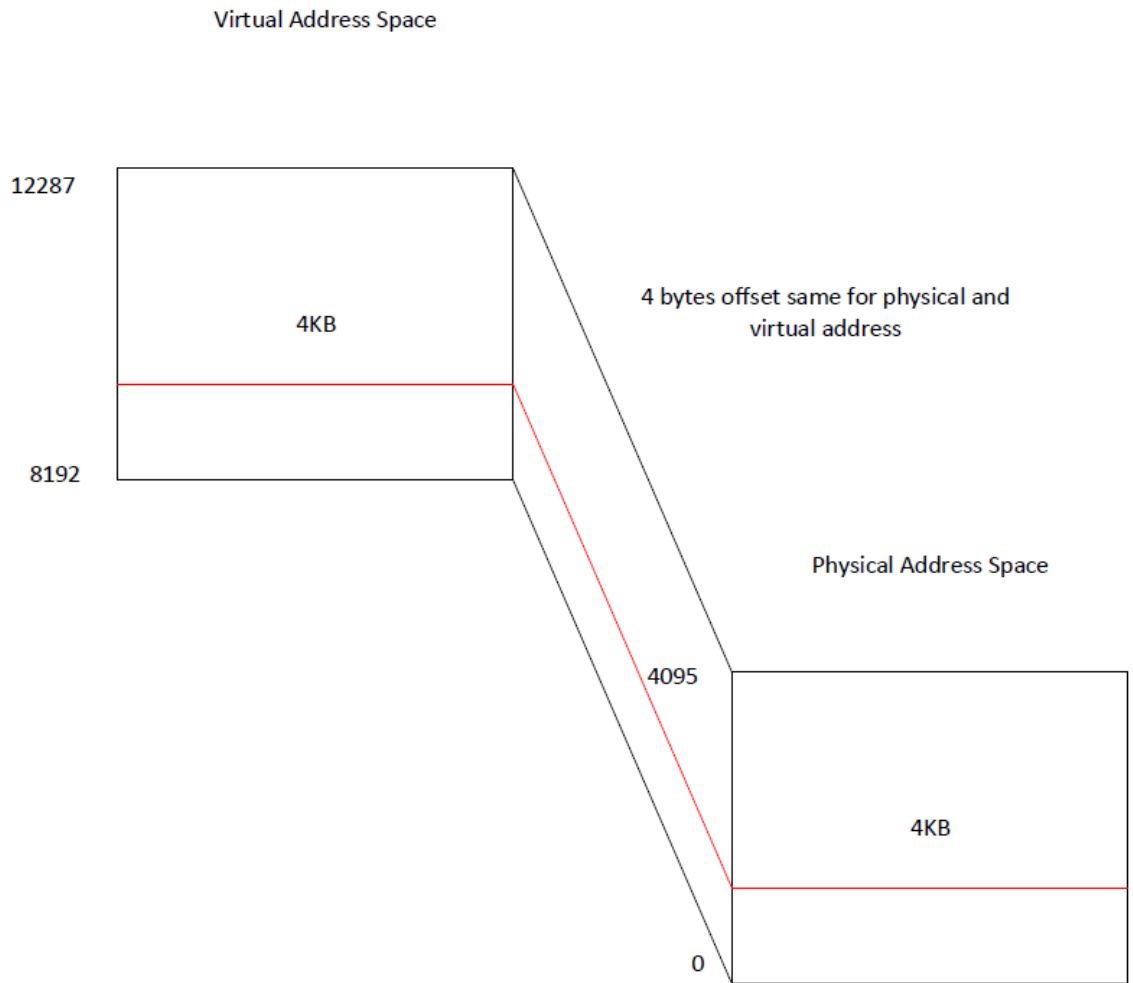
Figure 2. 4. Virtual Address Space

(Stazi et al., 2017)

For a 32-bit machine with a physical memory of 256 MB and 4KB page size, the offset

bits of 12 bits makes the page size 4KB (2^12 = 4K).

The virtual address is 20 bits which excludes the 12 offset bits. The physical address of

16 bits is formed for 256MB of physical memory, equivalent of 2^28 by excluding the 12

offset bits.

Offset bit 12 bits with a page size of 4KB (2^12 = 4K). Figure 2.5 illustrates the page map

table where the offset value remains the same.
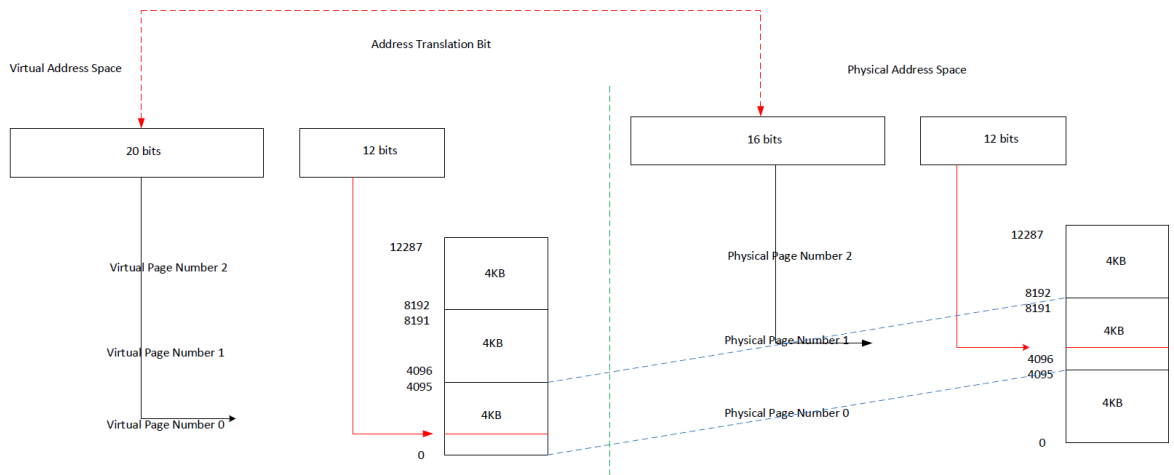
Figure 2. 5. *Virtual and Physical Pages*

(Ko et al., 2008)

Virtual memory is more than physical memory; hence the operating system must use the physical memory efficiently. It is done by only loading the required data in the memory. This way, physical memory is utilized efficiently, and the technique is called demand paging (Nothaas et al., 2019).

The operating system marked it as a page fault when the process attempts to access a virtual address, which is currently not available in the physical memory and hence processor could not find the page table entry in the mapping table for that virtual address. In such a scenario, the operating system may end processing the program as it may consider it a rogue application that might be accessing the system in the wrong way (Kim et al., 2014). On the other hand, if the virtual address is valid and does not have a valid physical memory address, it may look for the address on the disk. Due to the slow nature of the disk, it takes more time to fetch data and hence take more time to execute the process. If it has another job during the waiting period, execute that program to avoid any idle period. Once the access is made from the disk, the content is added to the free available physical address, and the map table is updated. Next time when the virtual address is called, it can directly access it from the physical address as updated in the map table (Chen et al., 2016). Linux uses demand paging to load executable images into

processes virtual memory. Whenever a command is executed, the file containing it is opened, and its contents are mapped into the process virtual memory. It is done by modifying the data structures describing this processes memory map and is known as memory mapping. However, only the first part of the image is brought into physical memory. The rest of the image is left on the disk. As the image executes, it generates page faults, and Linux uses the processes memory map to determine which parts of the image to bring into memory for execution (Chen et al., 2016).

When there is no space available on the physical memory, the operating system must make space for the new page by removing the page from the physical memory. If the discarded page in the physical memory is from an image or a data file that does not need to be written, the page is not saved. Moreover, if it again requires the image and data file, it can be brought again to memory from the disk. In case of page modification, the operating system must preserve the data to access later. The modified pages are referred to as dirty pages (Yildiz Cavdar et al., 2019). These modified files are temporarily removed from the memory and stored in a particular file called the swap file. As these swap files are stored on the disk, it takes a longer time to process. Its operating system manages the swap file and content in the memory to avoid long access time for the active files. If the algorithm to operate swap files is non-efficient, like moving the swap file on disk and the physical memory, it keeps the processor busy (Liu et al., 2016).

Any file which requires more regular access is not the ideal candidate for swap files. The set of pages that a process is currently using is called the working set. An efficient swap scheme would make sure that all processes have a working set in physical memory. Linux operating system uses the Least Recently Used (LRU) technique for page aging. It is to keep track of which pages may need to be removed from the memory. The age of the page depends on the time it is accessed. The age of the page decreases when it is accessed and increases when not. The old-age page file has more probably to move into the swap file (Chen et al., 2016).

Different processes use different map tables to maintain the translation between virtual and physical addresses, and some processes share the same physical address. In this case, two different process map tables with different virtual addresses point to the same physical address.

The Linux operating system runs on the physical address mode. Physical address mode does not require any page table, and the processor does not look for the map table to do the address translation.

The memory access to look for address translation is prolonged as compared to CPU speed. To overcome the waiting time for the CPU process cycle, the CPU maintains a cache of translation known as Translation Lookaside Buffer (TLB). Even though TLB is not efficient if the TLB cycle is missed, to overcome this modern inefficiency CPU, allow mapping the memory pages directly at the high hierarchy level in the page table. Such pages in Linux operating system is called massive, and it helps improve the TLB hit rate and enhance system performances.

Access control is also applied to ensure that the process does not take any other action that it does not intend to.

The physical memory can be divided into zones and allocations depending on the memory system used by I/O hardware and the processing unit. There are four categories of zones; the first one is ZONE_DMA, which uses lower 16MB of the physical memory. The second zone uses 32 bits address and is known as ZONE_DMA32. Due to the limit of 32bits address, it has a limit of 4GB. On x86 intel, the first 896MB, which is the direct memory access by the kernel, is referred to as the LOW_MEM zone, which the kernel uses for logical addressing. This region uses the "k-malloc" and "k-free" library functions for requesting and deallocating the kernel memory in this zone. This region also has a mapping to the physical memory and hence does not use the swap. The high memory zone is from 1GB to 4GB address region on a 32-bit system. This zone is only available

on the 32 bits system and 64, but systems only low memory zone is applicable figure 2.6.
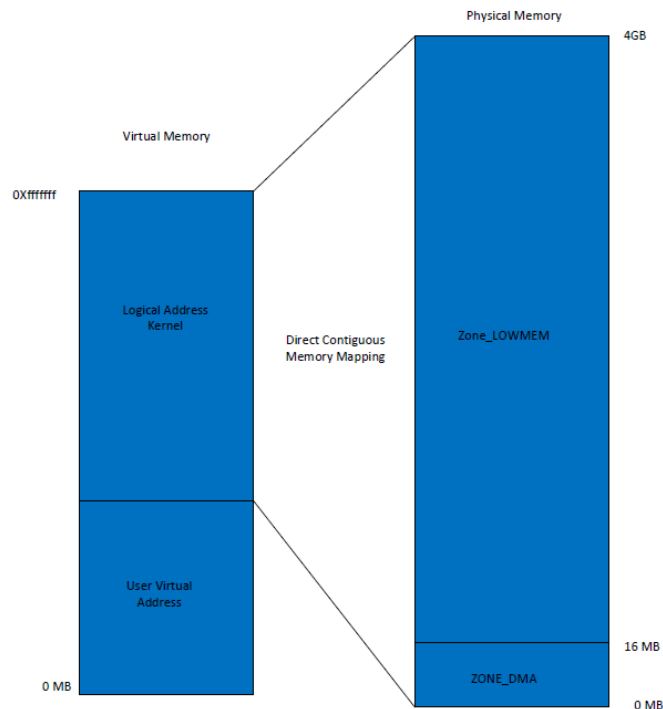
Figure 2. 6. *Memory Zones*

(Wu et al., 2012)

The pointer uses the address range from 896MB to 1 GB is reserved in 32 bits systems to reference the memory address in areas beyond the kernel logical address.

The address allocation/deallocation outside the LOW_MEM zone is done through "v-malloc" and "v-free" library functions. The difference between "v-malloc" and "k-malloc" is that "v-malloc" does not need to have contiguous memory in the virtual address and physical address mapping. It also uses more entries in the TLB because of non-contiguous allocation. The fragmentation is reduced using the memory zone allocator. The user-mode process uses the buddy system, where it keeps track of the adjacent memory block or splits the memory block by a power of 2. In this case, if a request requires additional blocks, the buddy system identifies the adjacent blocks that can handle that big request and place it accordingly to avoid the fragmentation or splitting of

large block spread in multiple blocks. On the other side, if the size is small, it repeatedly reduces the size of the block by two till it gets to the defined lower limit, which can handle the request.

2.4.2.2 Process Management

This section will brief the kernel process management. The systems call and signals management is essential in identifying any process initiated by the malicious software. In this section, scheduling, task and kernel synchronization is discussed

2.4.2.2.1 Signal Management

Process ID (PID) is a unique identifier assigned to each process. There is a fork system call that is used by the process to create another process in Linux. The process that calls or creates the process is called the parent process, and the new process is known as the child process. It allows the image in the memory and variables associated with the parent process to be called into the child process, enabling the child process to access the open files used by the parent process (Tanenbaum & Bos, 2014). Multiple child processes can be associated with the parent process, whereas the child process has only a single parent process. When the fork() call is executed if successful, the PID of the child is returned in the parent process, and 0 is returned in the child process. Due to any process failure in the call, the parent process gets the value -1, and the child process is not created. The child process generates a SIGCHLD signal to the parent process if its interrupts or exits the process. The only condition when a child process has no parent process is if the kernel creates it. The child process can spawn more child processes which can create a complex process tree; hence "get-PID" system call is used by the child process to get the process identifier of the parent process (Kerrisk, 2010).

The exec system call is used to execute a file that is residing on the active process. When this is called, a file is executed, which replaces the previous executable. Therefore, when the shell command is launched, it replaces the image of the environment's memory

and variable with the values that initiated the process. The WAITPID system call is used if it is required to suspend the calling process. This suspension of the child process remains till the child process state is changed. By default, if no argument is given, the suspended process remains suspended until the child process returns the terminated process identifier, which is -1. If the child process is completed but does not send the exit status to the parent process, then the WAITPID system calls initiated by the parent process will remain in the process table. Such open processes are termed Zombie processes (Love, 2010).

As the process identifiers are unique, multiple processes send the communication signal among the processes. The process which belongs to the same process group, also referred to as the process family tree, can send signals to another process in the same group. Signals can be used for multiple instructions like SIGHUP (terminate gracefully), SIGTERM (terminate unconditionally and immediately), SIGKILL (suspend itself) (Shotts Jr, 2012).

SIGACTION system call is used to change the action taken by a process by the receipt of a specific signal. To kill the process immediately, SIGKILL is used and does not wait like SIGTERM. It is not handled by SIGACTION and directly goes to the kernel (Tanenbaum & Bos, 2014).

2.4.2.2.2 Process Representation

The task_struck data structure represents each process in the Linux system. The task _struck has the array of pointers known as task vectors. The necessary process information like state, memory-related information, files, process details for parent and child processes are found in this structure. The kernel manages these pointers, and all the active processes are double linked in the task_struck. Process identifiers are used as the critical value of the task_struck (Tanenbaum & Bos, 2014). The size of the task vector also represents the allowed number of maximum processes in the system. As processes are created, a new task_struct is allocated from system memory and added

into the task vector. The current pointer points to the currently running process to make it easy to find.

When the parent process creates the child process, both these processes share the same pages in the memory. Shared pages are marked as "copy on write", which creates the page copy before writing to the process. It protects the pages modification, which could impact another process. It also helps in reducing the memory overhead and requirement during the creation of the process. It also improves the efficiency as the child process may not use the resources of the parent process, which may be terminated after spawn or may call another program which replaces its memory image and pages (Love, 2010).

2.4.2.2.3 Thread and Tasks

The research is based on detecting the malware on the abnormal behaviour due to the unusual process. Each process has a basic unit of execution which is known as thread (Fox, 2014). It implies that each process has multiple threads. It is essential to understand the components of threads to identify and stop them if it finds malicious activity. This capability can enable the run-time analysis of the behaviour of the program being executed successfully. The information in the kernel structure of a process can differentiate between malicious and benign processes (Shahzad et al., 2011).

The task structure as maintained by the kernel of the operating system contains has the record of each action and the amount of resource used by the process; hence this pattern of records must be different for the malicious process and benign process (Shahzad et al., 2011). The challenge for detecting malware on the processor threads is the accuracy rate. Hence modelling of such threads is essential to reduce the false alarm.

2.5. Related Work

This section discusses existing work done related to the detection and analysis of malware. This section also contains a few sandbox methods and reviews some sample malware on the Linux system.

2.5.1 Malware Analysis Methods

The following section reviews the current literature available on numerous methods for malware analysis.

2.5.1.1 Basic Malware Analysis Method

Malware analysis is a method to examine the malware component, its behaviour, and if an attacker can be identified. Figure 2.7 shows the flow of the basic malware analysis method.
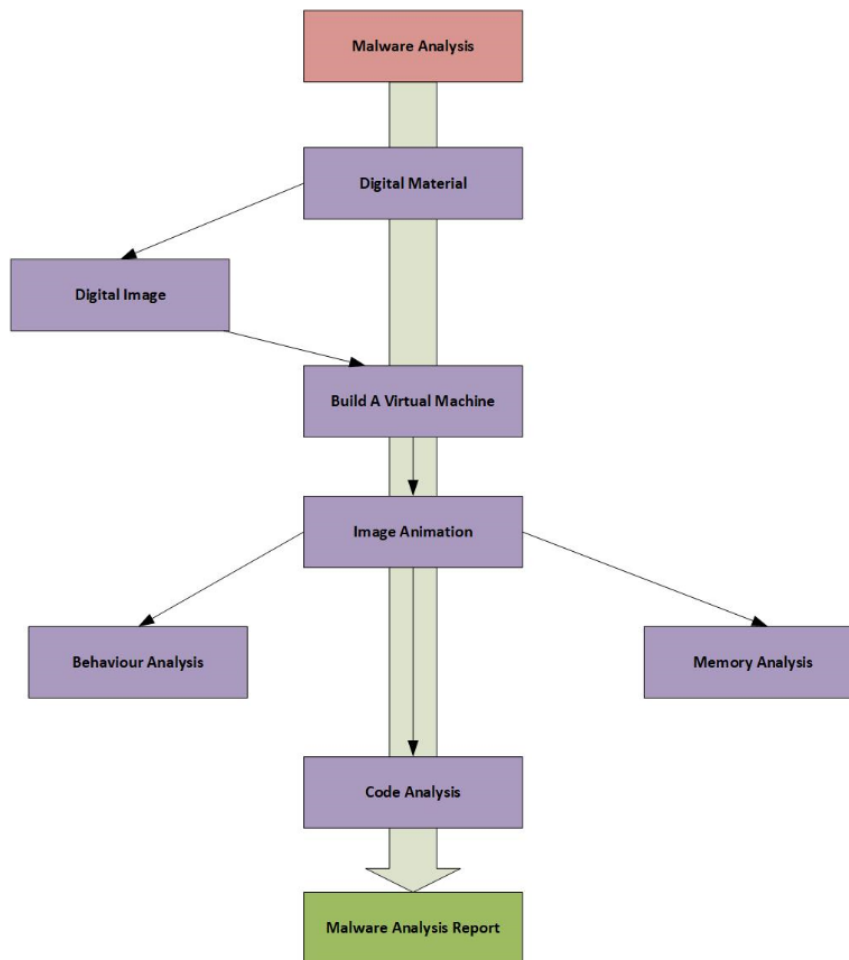
Figure 2. 7. Malware Analysis

(Kara, 2019)

Investigation of all analysis is done on the image (copy), and the hash method is used to detect the integrity failure. The image includes all currently available data, deleted data or any other data available in the storage. Image processing is the fundamental step for the analysis (Kara, 2019).

A dedicated and isolated virtual machine is created to mount the copy of the image to analyse the behaviour.

The behaviour analysis of malware includes registry activity, network operations, file and directory transfers. Sometimes unintentional bugs are introduced when creating malware. Hence, debugging can be performed with behaviour analysis. This error information is essential. In some cases, debugging can provide information about the attacker (Kara, 2019).

The code analysis is performed with static, dynamic or reviewing the packaging techniques. The static analysis includes viewing the text, function used, file directory, compressed file status, hash values and since it is activated in the system. Many attackers compressed the malware file to escape the detection from antivirus tools; hence, it is critical to understand malware packaging techniques (Vasilescu et al., 2014). Finally, the code is executed in a controlled and isolated environment to observe the activities. Attackers implement concealment methods (such as anti-sandbox, anti-VM and anti-debug) to prevent malware from being detected by antivirus programs in the system before its execution. Therefore, the data obtained from running malware is more critical than static analysis (Christodorescu et al., 2007).

Memory analysis is critical as processes in computer systems need to be loaded into memory to be executed. Even though malware might use different methods, such as hiding in the memory of other running processes, they cannot be caught by antivirus programs but still need to be loaded into the system.

All the activities of different processes working in the system will be in the memory. Therefore, memory analysis allows a better option to detect all the current activities. The current state of the system can be saved using the snapshot feature of the virtual machine. In the memory analysis, the list of processes running in the system at the time of the memory dump, the active or previous network connections, .dll files or codes injected into the memory areas of the processes can be dumped.

Malicious software hides from conventional detection methods and, in some cases, can change the way it behaves to deceive analysts. However, even in this case, they are still resident in the memory.

However, the memory dump contains only a snapshot of the computer's state. It is necessary to take continuous dumps to understand what has changed after the memory dump. It is limited by the time available for analysis. The memory dump can be retrieved on the live system, but this poses several risks. If malware is present in the system, it may intervene in

this process. Data in memory is volatile. Hence make sure that the system is not shut down as it may lose the data in the volatile memory (Gandotra et al., 2017).

2.5.1.2 Analysis System Components

The Two components provided by (Wagner et al., 2015) for malware analysis systems are data provider and analysis environment. The data provider includes the toolset or packages used for dynamic or static analysis of the malware samples. The Static tools like IDA, GDB, Radare2 and others, whereas Rekall volatility, are used for dynamic analysis. The environment to do the necessary analysis are tools like the cuckoo, threat analyser and others. A dedicated or isolated environment is essential for the dynamic analysis, set up on the virtual machine, bare metal or emulated environment (Wagner et al., 2015).

Machine learning libraries can be used as well to extend the capability of malware analysis. It helps in the learning system for the classification and detection of malware. The output collected from the data provider is used with the machine learning algorithm to build a more robust system (Shah & Singh, 2016)

(Boukhtouta et al., 2016) in his trial utilised the dynamic examination of malware tests and documents to prepare the frameworks for mechanised malware recognition and characterisation. In (Shah & Singh, 2016), the extraction of prominent API calls from benign and malicious files was used as input to linear support vector machine (SVM).

(Boukhtouta et al., 2016) using the Threat track online sandbox for deep packet analysis and examination of flow packet headers in malicious network traffic, trained a system using ML algorithm. Benign traffic was sought from the Internet service provider edge and customer traffic.

(Shah & Singh, 2016) did the malware analysis use multiple modes? Modes include user mode, kernel mode, full system simulation or emulation, virtual machines and others. API is used to call to collect the relevant information for the data.

2.5.2 Indicators of Compromise

As per the National Institute of Standards and Technology (NIST) publication on Intrusion detection and prevention, one of the primary purposes of using IDS and IPS is to detect attacks (Kizza & Migga Kizza, 2011) and other security breaches which cannot be detected or prevented with the usual security methods. Abuse detection and anomaly detections are the two standard methods to analyse the detection. In abuse detection, if any attack occurs with the pre-defined method or warning, then such characteristic of abuse detection is an indicator of compromise (IoC). One of the methods to determine the IoC is to do the malware analysis (Akram & Ogi, 2020)

IoC can be determined by using the reverse engineering technique while doing the malware analysis. It is a forensic artefact of an attack that exist either on the machine or on the network. The data collected during the malware analysis is termed the artefact. It may include files, URL, IP, process, and registrations that would have been used to execute malware. The artefact collected is used to set the indicator of compromise. All the data collected during malware analysis is not included in the indicator of compromise. Artefacts are classified into two categories which are network and host-based artefacts. Data collected from the network includes ports, servers, and proxies used. Tools like packet capture are used to collect this data. The host-based can include the files, memory, or process usage (Boukhtouta et al., 2016).

Reverse engineering for malware requires a disassembler that converts code to the assembly language. The other component is the decompilers which convert machine language to the high level for the easy understanding of engineers doing the reverse engineering for the malware analysis to find the indicator of compromise (Mohanta et al., 2020).

2.5.3 YARA

The rapid growth of malware incidents indicates a problem in malware analysis. This pace of malware growth poses a challenge for malware analysts. There are multiple techniques available for malware analysis concerning specific security incidents. In present times YARA rules have come up as the widely accepted technique for malware analysis due to their flexibility and customizable nature. As per (Naik, Jenkins, Cooke, et al., 2020) It allows the analyst to create YARA rules as per the specific requirements to handle the specific threat. It is also based on the reverse engineering technique of different malware to include the standard indicator of compromise strings from those samples and use this data to find similar types of malware.

The effectiveness of YARA rules determines which types of IoC strings and the numbers of these strings in the rule. Therefore, the biggest challenge is to create an effective YARA rule for malware analysis. These rules can be created either manually or automatically. The manual creation of YARA rules requires much skill in specific security areas. The automatic rules generations can be created using specific tools hence relatively easy. The challenge with the automatic method is the optimization of requirements and less effectiveness with multiple threats (Naik, Jenkins, Cooke, et al., 2020). When YARA rules are triggered, it generates a malware alert if the malware matches the string condition. If the sample condition does not match, it does not trigger an alert and requires the rule to be updated. Note that the malware string is created using IoC from the sample, which means the classification of the malware being analyzed is not the same as the sample. These issues can be remedied by adding more strings; however, size increases adversely impact the YARA rules performance. Secondly, adding or modifying the string needs expertise in computer security (Naik, Jenkins, Savage, et al., 2020).

Multiple open-source tools are available: python-based like YarGen, YARA generator, Yabin, and others. Fuzzy hashing aided is one of the techniques available to improve

the effectiveness of the YARA rules without adding complexities. The hashing method is applied alongside YARA rules which complement each other. Hence if the malware alert is not triggered by one, the other method can detect it.

The Fuzzy hash method is used to find the similarity of the digital files. This assists in making a valuable method for malware analysis because many malware and variants contain similarities. It is different from the cryptographic hash as the variant will make the hash of the two files differ. In the fuzzy hashing technique, the file to analyze is divided into chunks, and it calculates the hash against each block. Then these individual hashes are merged to form a concatenated hash called the fuzzy hash of a file. The fuzzy hash value depends on the size of the block, the size of the file and the output size of the hash method used. The malware analysis needs a deep knowledge of the similarity of the known sample malware and the file being assessed for malware. It becomes crucial when work is done to identify the variant.

2.5.4 Sample Malware Analysis on Linux System

Many machine learning methods were used to detect malware in Linux systems (Mehdi, S.B., Tanwani, A.K., Farooq, 2009) and (Shahzad et al., 2013). Different structures were used for these detections like (Mehdi, S.B., Tanwani, A.K., Farooq, 2009) used task struct for the classification, whereas (Shahzad et al., 2013) worked on the ELF file structure. The (Mehdi, S.B., Tanwani, A.K., Farooq, 2009) technique was able to get 99% accuracy using multiple malware samples, which is an excellent method to detect zero-day malware. As per (Shahzad et al., 2013), the used system calls when ELF files were executed, which produced 96% of accuracy.

(Damri & Vidyarthi, 2016) used dynamic analysis of malware to identify five approaches. These uses system call, process control block, ELF, kernel and hybrid investigation, which uses four methods. Different tools were used for the investigation of the system calls. Some components of the ELF header were used in the research to identify the

difference between benign and malicious programs. Proc and memory files were used to collect the kernel state to determine any malicious activity by the program.

(Cozzi et al., 2018) used the sample collected from the virustotal. They identified the challenge associated with the behaviour of malware with different CPU architecture, different obfuscation techniques. Dynamic linking and shared libraries were also highlighted as one of the challenges for malware detection.

2.5.5 Zero Day Malware Detection

Zero-day malware is an unknown vulnerability that has been exploited and has not been detected by anti-malware tools (AMT's) (Ciancioso et al., 2018). The expanding volume and assortment of malware postures a genuine security danger to all the computer these days and is one of the most worries for the security community for the final few a long time (Or-Meir et al., 2019). The conventional security frameworks like Interruption Location System/Intrusion Avoidance Framework and Anti-Virus (AV) computer programs cannot identify unknown malware as they utilize signatures-based strategies. In arrange to illuminate this issue, static and dynamic malware analysis is being utilized in conjunction with machine learning calculations for malware discovery and classification. The biggest issue with these frameworks is that they have elevated wrong favourable and wrong negative rates, and the method of building classification demonstrated takes time (due to colossal highlight set) which ruins the early discovery of malware (Gandotra et al., 2017). Hence, the challenge is to choose a significant set of highlights so that the classification show can be built in less time with more accuracy.

Due to the pros and cons of both static and dynamic malware analysis approaches, it is apparent to know about malware classification. Besides large numbers of false positives, the classification modelling takes a long time and thus preventing the early discovery of malware. Hence, a pertinent set of features should be selected to build the classification in less time with more exactness. Feature choice could be a strategy of recognizing best-

positioned highlights. It identifies the critical features hence making it simple to dispose of the insignificant ones.

The anti-malware tool relies on the malware database and, as per the stats, misses one-third of malware without having prevention measures (Ciancioso et al., 2018). A method that combines rule and algorithm are also challenging to identify zero-day malware. Such methods are known as heuristic-based scans, and these methods do not rely on signatures. As this method also relies on finding a specific code piece, such a method is not adequate for future and evolving malware. If the inquiry does not contain the instruction and is unable to read the file or code due to encryption, the malware will be undetected. There is complex malware that can remain dormant for a specific time before conducting any damaging work. These days most anti-virus programs use both methods, including traditional signature matching and the heuristic-based, to detect the malware (Ciancioso et al., 2018).

Sandboxing method is also used, which keeps the malicious code in the contained area. It is also termed a virtual cage that prevents malware from infecting the existing operating system. The detection of malware activities detected in the sandbox environment is kept in a centralized database which assists in identifying the variant of the malware. Hence an application is designed to automatically create a new variant of the malware based on the results of previous malware. It helps in detecting zero-day malware. Hence the more variants of malware created in a sandbox increase the possibility of zero-day malware detection (Ciancioso et al., 2018). Sandbox alone cannot provide the complete foolproof solution for zero-day malware detection; hence, it needs to use other invasion methods like stalling code and blind spots. The stalling code runs after the timeout period and hence help in detecting the malware, which keeps dormant for some time. Sandbox also introduces a method called hooking. In this, a notification is generated whenever any function or library is called. The challenge with this method is that modification is required every time to make it effective.

Furthermore, to zero-day detection, there is malware that has evolving capabilities such as polymorphic and metamorphic. Polymorphic malware is difficult to be detected as they mutate by themselves and also uses encryption techniques has cause challenges in being detected. Metamorphic malware can reprogram automatically each time they are executed or spread in the system. Hence in such categories, signatures will not be detected or, due to encryption, the payload cannot be read to detect the malware (Comar et al., 2013).

A supervised classification also uses the known instance of the malware and set the classification. It has the same limitation as to the signature-based approach for detecting the malware as both cannot detect the new or evolving malware. Also, making a classification table is challenging due to diverse malware classes, imbalanced distribution and data loss issues. The unsupervised method uses the anomaly detection technique. The significant benefit of this method is the ability to detect zero-day attacks. The challenge with this method is the high number of false positives. Hence white listening must be applied for the wrongly identified applications (Comar et al., 2013).

Network traffic flow can also detect malware behaviour on layer 3 / layer 4 of the OSI layer. This method is based on the traffic features such as bytes per second, packets per-flow and inter-arrival times. The approach is that the statistical flow level features at level 3 or 4 remain the same even if the payload is encrypted. The challenge with this method is that the malware must be detected during the network flow. It also possesses the challenge of any new zero-day malware which is not detected as an anomaly.

The rise of artificial intelligence has also been involved in the cybersecurity space to detect malware and reduce false positives. One of the studies proposed a custom log loss function with beta parameters to the GDBT algorithm to solve the malware detection problem. Using a rational approach to evaluate the proposal, they extracted 27 valuable features from the PE surface analysis FFRI data set. The result shows that the custom log loss function can reduce many false positives than a normal log loss function.

However, the custom log loss function increases false negatives compared to the standard log loss function, so a hybrid model was used to keep the balance. The reduction of FPR value with custom log loss function could reduce the priority of false positives (Gao et al., 2021).

2.6 Research Questions

Zero-day malware poses a considerable risk as signatures are not available for the anti-virus tools to block or remove the malicious files (Gandotra et al., 2017), as elaborated in the literature review. Multiple techniques are available which requires sandboxing and include dynamic or static malware file analysis techniques. Sandbox technique executes the code in the isolated environment to analyse the behaviour of the files and, based on it, marks it as malicious (Vasilescu et al., 2014). The need is to design a solution to identify the malicious files and create a process that shows anomaly behaviour.

Hence the research is seeking to answer the following questions:

Question 1 (Q1) – Which system process on Linux system can detect anomaly behaviour?

Understanding the tactics, techniques and procedures of the malicious file to perform malware analysis requires a sandbox to analyse the pattern of the file execution (Li & Liu, 2017). This method identifies the malicious file, which creates the signature for the file. The signature update is shared with the threat intelligence available publicly or privately through paid subscriptions, leading to multiple anti-viruses to update the database to block the file in the future. The current signature-based approach will not be able to detect exploitation by the malicious software for which signatures are not available, and hence a process is needed to detect the zero-day malware. One source of information that detects the malicious file when it executes is the abnormal activity detected by active processes. Hence there is a need to create a flow process in which

the system process is monitored to detect such anomaly behaviour to identify the malware and can be blocked on the run time.

Question 2 (Q2)- How can YARA rules and antivirus software be integrated after the zero-day malware is detected?

After the process is detected to identify the malicious file, the next step is to update the endpoint signature. The most common technique is to triage a malware and then separate the likely and unlikely malware. YARA rules are one of the everyday use of such triaging techniques. It identifies using the pattern or string matching, which triggers the rule on the matching condition. There are multiple methods available for the YARA rules detection to reduce the false positive, and one of them is fuzzy hashing (Naik, Jenkins, Savage, et al., 2020). There is a need to make a process that updates the malware string and pattern to update the anti-virus signature on the endpoint.

The literature review leads to a concern to detect zero-day malware analysis, and the following hypothesis has been generated.

Hypothesis (H1):

Abnormal activities on the Linux operating system with the combination of unusual processes and the number of times it repeats is used to detect the anomaly behaviour of the malicious software exploitation.

The detection method of the malicious file can be used to integrate YARA rules and endpoint antivirus tool to update the signature to detect or block the zero-day malware.

.

3.   Research Design

This chapter discusses the design methodology and the purpose of the design decisions. To analyse the malware for zero-day detection, it needs to acquire data samples for the multiple processes for testing purposes (Jicha et al., 2016). This chapter covers the technique used for collecting the data for zero-day malware detection, and the research objective is discussed. This section highlights the method which is used for the detection of zero-day malware. It also highlights the role of individual components used in the setup.

3.1 Research Approach

The following section discusses the approach for setting up the environment to collect the data for the investigation to determine the answer to the research questions. The approach includes collecting the sample malware, briefing the suitable setup environment and the analysis techniques.

3.1.1 Sample Malware

Malware researchers look for the sample malware to analyse the threat. One of the sources is collecting samples of the malware available in the public or private repositories. Researchers, anti-malware vendors and different security vendors can share the threat intelligence through multiple public available portals. Malware is added in the portal, which is performed a cryptographic fuzzy hash function to identify if the newly added malware is the same as the previously added malware. Few available repositories are free to download malware; however, lots of them require some registration. Few available repositories include (Zeltser, 2021), (Shipp, 2020) and (Virus-Share, 2020).

Virus Share has more than 38,000 malware samples available (Virus-Share, 2020) in the repository, which can be used for training or demonstration purposes. Here the significance is of detecting zero-day malware attacks on the Linux operating system rather than the number of the malware. The other approach to acquire the sample malware is the use of honeypots. In this approach, a dedicated system is configured to attract the attacker, which gives a good understanding of the recent attacks using the malware (Guarnizo et al., 2017). The classification of these malware depends upon the interaction permitted (Mairh, Barik, Verma, & Jena, 2011). The level of the malware is decided on the level of the interaction with the honeypots. This approach requires a sandbox technique after the malware is uploaded by the attacker on the honeypot. Sandbox provides the malware analysis of the file and helps to provide the details of the file, which help in writing the YARA rule.

3.1.2 Design Setup

The setup involves the following components as shown in the figure 3.1.
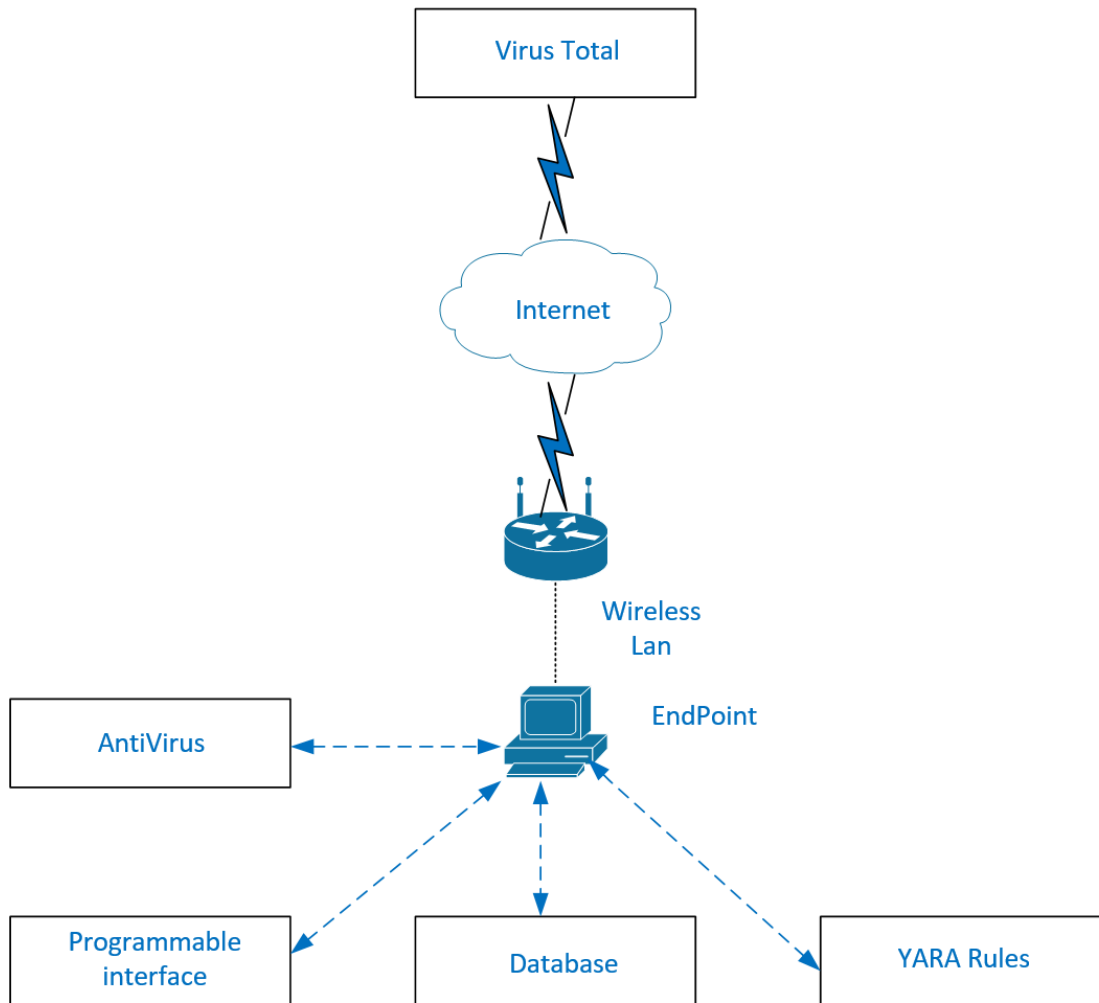
Figure 3. 1*. Design Setup*

The following section elaborates on the components involved in the Lab setup. It details each component's role in providing the answers to the research questions and testing the hypothesis. It provides the details for the components installed on the end-point. It also discusses the APIs which interconnects multiple segments of the Lab setup. The setup includes the role of the YARA rule and its integration with the end-point. After highlighting the roles of different segments, the section analytical method will provide the complete flow process by joining multiple roles throughout the process. At the end of the chapter summary of the design is presented in the concluding section of this chapter.

3.1.2.1 End Point Details

The Linux operating system has multiple different distributions available. For the lab setup, CentOS Linux distribution is used. This distribution of Linux is manageable, stable and predictable (CentOS-Org, 2021). Following table 3.1 shows the operating system and hardware specs of the test endpoint.

Table 3. 1. Endpoint Hardware Architecture

Name	Version/Model/Size
Centos Version	7.6.1810
CPU Architecture	x86_64
CPU Count	4
Threads per Core	2
Core per socket	2
Socket	1
Vendor ID	Genuine Intel
Model Name	Intel ® Core ™ I5-4300U CPU @ 1.90GHz
Memory	8GB

The endpoint is connected to the wireless network with the following local network details:

Table 3. 2. Wireless Network Details

Wireless Lan	wlo1
Endpoint IP	192.168.20.19
Mask	255.255.255.0
Gateway	192.168.20.1

The gateway is the device that provides dual services, including the wireless access point, and is also connected to the service provider that connects to the Internet.

The setup does not have a firewall in front of the Internet-bound traffic; therefore host-based firewall is enabled. The CentOS supports the FirewallD service to enable the firewall services. The firewall is critical for controlling the traffic flow in and out of the Linux machine on the network (Carrigan, 2020).

The setup is connected to a shared network hence not inherently safe while doing the malware testing; therefore, it is mandatory to enable the firewall services on the end-point host. Firewalld service provides multiple security levels for different zones. A zone must be connected to at least one network interface. The end-point in the setup comprises a single wireless LAN card; hence does not require multiple zoning. However, the network traffic is controlled by allowing and denying multiple services. Following firewall policies are applied on the end-point setup.

Table 3. 3. *Firewall Security Policy*

Source Zone	Source host	Destination Zone	Destination host / Network	Application	Policy
out	192.168.20.19	out	any	HTTP	Allow
out	192.168.20.19	out	any	HTTPS	Allow
out	192.168.20.19	out	192.168.20.1	DNS	Allow
out	any	in	192.168.20.19	all	deny

Only Internet traffic for HTTP and HTTPS traffic can access the Internet. The DNS is configured to access the local DNS server on the network. All the other inbound traffic is blocked, including remote management. These controls are strictly applied to avoid any compromised attack on the end-point while working on test malware. The deep packet inspection to check the header of the application is not applicable as the end-point encrypts and decrypts the HTTPS packet (Carrigan, 2020).

For the test setup, an open-source end-point antivirus is installed ClamAV AntiVirus. It is an open-source (GPLv2) antivirus toolkit that has been designed especially for scanning the emails on the mail gateway, but this antivirus engine provides sufficient end-point protection for the lab setup. The core engine for the antivirus is the use of the shared library to update the signature packages. The utilities include a scalable and flexible multi-thread daemon, which supports the command line to run the scanner. It is also equipped with advanced tools to update the databases automatically. It is designed to scan files, and the real-time protection option, which is only available for the Linux

system, made it the first choice to use in the setup. It blocks the file access until the scanning is completed (Clam-AntiVirus, 2020). This functionality is required to update the antivirus signature database. Due to such a feature, this open-source antivirus tool is used. The antivirus supports the detection of complex malware as it allows to use bytecode interpreter, which helps the writer of the ClamAV signatures enhance the remote scanner's functionality remotely. One of the reasons to choose ClamAV is that the platform is available to build and support CentOS7, and the following is the minimum requirement for the setup.

Minimum recommended RAM for ClamAV for Linux system includes 2GB of memory and 1 CPU with a minimum storage of 5GB.

ClamAV CVD and CLD database databases have been unpacked in the existing main installed directory. The database archives, including CVD and CLD databases, are configured to work with the custom database. CVD is referred to the database for the ClamAV, whereas CLD files are the uncompressed version of the CVD, which is not assigned. A custom database is required, which will be used by the interface to update the signature of the malicious file. The custom database exists at the location /usr/local/share/ClamAV. The default location is used to maintain simplicity. Otherwise, the alternate path is to be added to run the clamscan. The end-point configuration file is also tuned to work as per the setup environment. The file is referred to as DCONF (Dynamic Configuration). These settings are spread in multiple daily.cfg files. The categories which are opened for the test of files includes PE, ELF, DOCUMENT and PCRE. The other available sample files MACHO, ARCHIVE, MAIL, PHISHING, BYTECODE, and STATS are not used for testing purposes. ClamAV is also enabled to use the Magic number mechanism to determine the type of the file. The file daily.ftm on the end-point is adjusted to enable the File Type Magic Signature feature.

The custom signature created have used these naming conventions, -zipped suffix in the signature of the malware file for the file type zmd. For the file type rmd the suffix of -

rarpwd is used to name the malware signature. The system only allowed to use of alphanumeric. A dot (.), underscore (_), dash (-) and does not have any space, quote mark or apostrophe. (Clamav-signature-update, 2020). Another important file while writing the efficient signature file is the file containing the debug information from libclamav. The library is called by calling clamscan debug and leave-temps flags. CASC (ClamAV Signature Creator) tool is also used to perform reverse engineering in creating ClamAV signatures (The-Talos-Group-AT-Cisco, 2018). The product is not integrated into the setup however used for references. This product is a compatible IDA Pro plug-in.

HTML and Text files are also normalized using "sigtool" for the respective format. The sigtool is used to pull the libclamv and provides the clamscan, which executes at the back end.

3.1.2.2 Programming API

As shown in figure 3.2, different components are interconnected, including database, antivirus, Virustotal and YARA rules. These components are retrieving and sending data to interconnected components. Application Programming Interfaces are most commonly used to collect or send data. The setup is based on the language which is easily accessible and has readily available interfaces with different components involved in the setup. Python is used to perform these tasks, and the version used for the setup is 3.6.8. The zero-day malware detection for the setup is based on reading the active process and doing the necessary recording. For fetching the process details, Python has the built-in library known as "psutil". This utility provides the process and system utilities, and it is a cross-platform library for collecting the information for the running processes and the utilization of systems. The system utilization includes CPU, memory, disk, network, sensor. The utility also provides monitoring of the system, setting the profiling and limiting the process resources. The management of a running process is managed by this utility (Pypi-org, 2020). The setup is prepared to detect the zero-day malware based on

detecting the abnormal process; hence, the prime focus is on the process monitoring and management. The following essential component is to record the process details. A programming interface is required to update the database tables. Details for the database table can be referred to in section 3.3.2.3. Python library, which interfaces with MySQL, is "MySQLdb". This utility can write again the DB-API compliant MySQL module, which can increase performance. The other component that enables Python is to install the MYSQL Python driver (Engel, 2017). Authentication is required for the Pythonaccessing the database. For the lab setup, a local username and password are used. The integration to threat intelligence is to verify the signatures of the existing malware. The Python application programmable interface supports the virustotal. The library Python uses to integrate virustotal is "virustotal-API". The integration allows Python to verify the existing software malware signature using the cryptographic hash function. The integration of the programming interface has a critical component with the antivirus tool. In the setup, ClamAV integration with the Python API is used. The available library for antivirus is "pyClamd". The interface is used to detect the virus for the python-based software. The interface is used to update the local database signature of the antivirus on the end-point after the detection process, as detailed in section 3.3.3. Once the signature is updated, the malicious software is deleted or blocked for execution. To detect malware and file-based threats, ClamAV relies on the signatures to separate the clean and malicious files (Clamav-signature-update, 2020). ClamAV signatures are based on the text and have to be in a specific signature format. These formats are supported and written in the YARA format. The YARA rule is used to automate the signature update.

An API is created using Python to set up the interface. The library used for YARA from the Python is (Alvarez, 2021). It covers all the YARA features from compiling, saving and loading the rules from the scanning files, strings and processes (Alvarez, 2021). The library is "Yara-python", which is imported to update the YARA signature. The ClamAV collects the signatures in CVD (ClamAV virus Database) files (Clam-AntiVirus, 2020).

The CVD file format delivers a digitally signed container that encapsulates the signatures and confirms that any malicious third party cannot modify them. The signature in the setup is actively verified in the public available threat intelligence Talos intelligence.
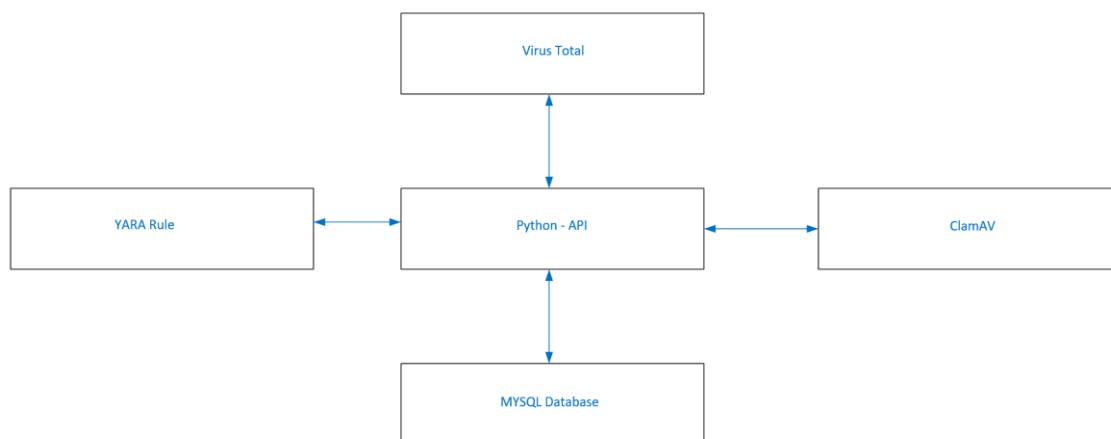
Figure 3. 2. Programmable Interface

3.1.2.3 Data Acquisition

The setup requires a database to save the process information of the operating system. The test environment is set up using an open-source version of MySQL under the general public license. MySql version 8 is installed on the end-point, which is currently supported on the OS version of the end-point. The network complexity is avoided by installing the database server on the same end-point. However, the methodology would work even with a separate machine. MySQL was available from the portal (MySQL-Dev, 2021). The package name is "mysql80-community-release-el7-3.noarch.rpm". For security purposes, default settings like changing the default root password, removing the anonymous user, disabling remote access for the root user as the DB is installed on the same end-point. The process to monitor the data acquisition are system, daemon and batch processes, interactive processes and zombie processes. Note that the system is the parent of all the processes and is managed by the systemctl.

Daemons are the process that runs continuously in the background and ends with the 'd' like httpd. Batch processes are in the spooler area, where it is executed on a first-in, first-out basis. The terminal sessions are initialized and control the interactive session. The

process which is closed but still possesses the process ID is referred to as the Zombie. Multiple commands are considered, including "top" (Table of Process) and "ps" (Process Status) to view the process ID (PID) of the active process. Though the top command runs continuously in the background, it is used as part of the Python script ps command with the flag involved "-aux". The flag provides the information for the user, process ID, the percentage of CPU and memory used, start time, and the command execution. The programming interface "psutil" is executed from python. The PS utility is the interactive command that is to be given each time the process is read from the system; therefore, a repetitive loop is configured. The selection of schedule is also important after successive testing. Keeping large intervals results in losing the critical process information, whereas keeping small intervals causes a large number of data. Hence 2 min interval is kept, which executes the PS utility and get the necessary update to populate the information in the database. The database is created with the table that contains the field for user detail, process id, CPU, memory, vs2, RSS, tty, stat, start, time, command, which includes all the fields output for the ps -aux output. The insert operation for MySQL database through Python needs to import the package for MySQL.connector package. After the package is imported, the first step is to establish the connection using MySQL.connector.connect() method. Username and password are required for the authentication, and for this setup, the same root account is used to access the database. The database name created is "processed", which is also called during the connection phase. A cursor object is created by calling the cursor() method with the connector details. The information is collected from the psutility. The output is inserted into the database by executing the cursor. Each time values are added to the table, a commit is required in the database.

. 3.1.2.4 VirusTotal Integration

Virustotal integration provides following two services for the setup

- Verify the signature of the files which is creating anomaly activity.

- Update the virustotal if the anomaly detection by the file execution is malicious.

The programmable interaction with the virustotal is through API version 3. Initially, the connectivity test was conducted via version 2, which was updated to version 3 due to functional limitations. The setup has used the premium version of the API integration, which allows the option to give the request rate and the daily quota of 100 requests per day. Signature files are downloaded from the virusshare site in the JSON format to validate the existing file. The virustotal hunting is included for the retrieval of the YARA rule notification. The integration has enabled performance property to sample queries along with the reverse searches. In the first use case for the setup, to upload the file libraries os.path and pprint are also imported. The OS path gives the ability to the Python to access the file structure of the operating system. The file which is to be tested for the signature validation is given to the API call. It returns the JSON response, which details the threat intelligence response from the virustotal accumulated from the known vendors marking as the threat. Another method used to validate the file is to only send the file ID in the form of the hash and collect the response in the same way as uploading the file. The hash values such as MD5, SHA1, SHA256 and the fuzzy hash of the samples can be uploaded. The second use case is to report the file to the virustotal portal, approved by the admin to be added to the threat intelligence portal. The submitted sample file to virustotal automatically go to the sandbox environment, and after doing the analyses, a verdict is assigned to the file with undetected or suspicious (Virus-total, 2021). The sample submission date indicates the life cycle of the selected malware. The addition of date gives a good indication of the age of the reported malicious software. VirusTotal is also used as a threat hunting in this setup, which differs from standard threat management measures where the approach is reactive than threat hunting (Alvarez, 2021). The report generated by the virustotal has the details with the finding like undetected, Suspicious, unable to process the file and timeout. The undetected refers to the fact that the engine cannot detect any malicious content in the file. The suspicious

behaviour raises the flag that the uploaded file is malicious. If the file is not able to recognize, then it returns the unable to process response. In unable to process and Timeout responses virustotal unable to make any verdict (Virus-total, 2021).

3.1.2.5 YARA Rules Update

After detecting zero-day malware, the setup is required to answer one of the questions to integrate the YARA rules and antivirus software, ClamAV. YARA version 3.11.0 is used for setting up the test environment. This section will discuss the YARA integration with the programming API and the signature update procedure for the ClamAV. YARA is accessed from Python using the Yara-Python library. To support the YARA feature for the SSL OpenSSL library is also installed. The Python API is used to verify if the unknown malicious instance causing anomaly behaviour belongs to a known malware family or is a novel malware. It also verifies the behaviour feature used to differentiate the instance of one malware family from those of other families. To differentiate the existing strain with the zero-day malware detection listed, the list of existing malware types includes Trojan, Rootkit, Backdoor, Xfil, Worms, Ransomware and keylogger. It allows the classification of the database to identify the category and help mark the category as zero-day. YARA rules for the existing strain are referred to the GitHub library for the YARA rules (Yararules, 2021). For the setup, the list of existing malware types is downloaded from the respective git-hub repository and for the initial test manual YARA command was executed to verify the detection of the random dummy files with and without sample malware. The exercise is mandatory to validate the impact of the false positive for the YARA signature. The sample malware and the YARA rules for the signatures are presented in chapter 4. This validation was also required to justify the load on the end-point as YARA rule verification is also performed on the same end-point, which has SQL database and the Python API interfaces. For the setup, Python is used to call YARA libraries to match the identified file with the existing malware families. If the signatures are not identified, and the file is not classified as malware from the existing matching rules Python interface is called again to create a YARA rule by analysing the file. The file

analysis identifies the critical string to match the signature of the file. Once the signature matching patterns are identified Python interface is called again to create the file for the matching YARA rule. The last step for YARA rule integration is to update the ClamAV signatures. After the rule is created, the ClamAV scan is reinitiated to delete the malicious file. A new file is created with the corresponding signature for the specific signature type, e.g. .ldb extension for the logical signatures. Then a new own line is added for the new signature in the same file. Clamscan is called using the Python interface to load the newly added signature. The clamscan is called again using the Python interface, which updates the local signature database. The format for the signature is essential as, during the testing, it will throw an error with the display error in the debug file output.

3.1.3 Analysis Method

This section discusses the analysis method used to evaluate if the process flow of the zero-day malware detection and explains the integration of individual components as mentioned in the sections on the Linux systems against the sample Linux malware. The zero-day malware detection process is evaluated against the malware sample collected from the repositories at virustotal and virus share.

Figure 3.3 illustrates the steps in detecting the zero-day malware process.
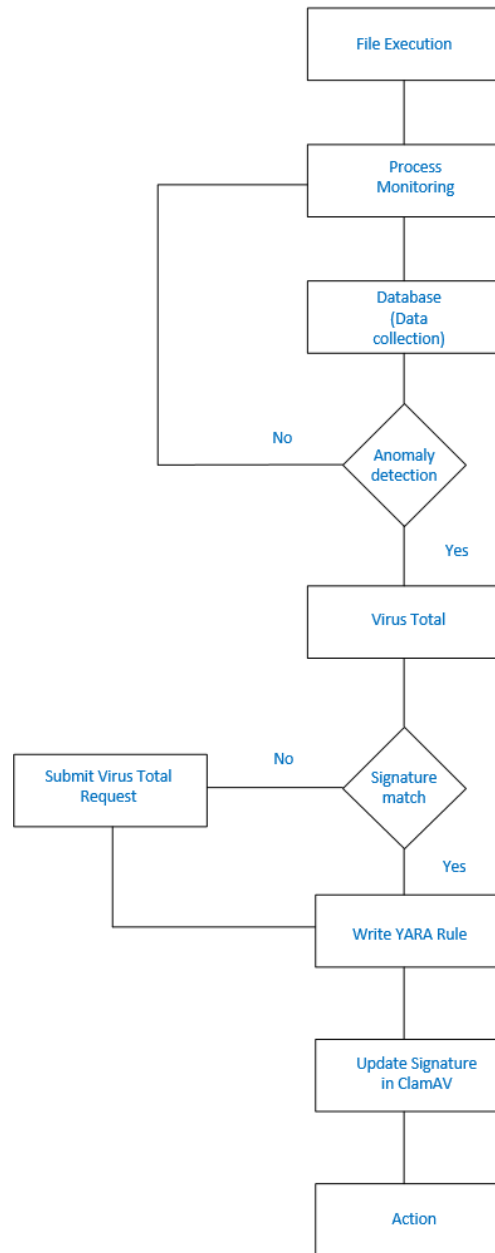
Figure 3. 3. *Process Flow*

The process flow was tested to detect the malware behaviour technique by monitoring the malicious activity on the end-point. For the analysis of the process for zero-day detection, some malware signatures were deliberately removed from the signature database of the ClamAV tool. The malware was known to be the malicious file, but the action was taken to verify the effectiveness of the process and to validate the false negative of the file detection.

3.1.3.1 Sample files

Sample malware was downloaded from the virus share and virustotal repositories. These samples were first downloaded on a sandbox machine with only an Internet connection. The files were executed to analyse the behaviour of the malicious file. There are multiple sandbox solutions available. However, for the setup, the only cuckoo sandbox was used. Cuckoo is also based on the Python library that executes the test sequentially for the analysed malware. Its integration with open-source tools like YARA, virustotal, volatility and support on multiple virtual machine platforms like VirtualBox made it the only choice for this setup.

The identified malware are verified with the YARA rules configuration and the available signatures in the ClamAV antivirus. For the setup, the signatures of a few malware files were removed from the ClamAV database. This action is required to make the signature of these malicious files unavailable so that the traditional antivirus is unable to detect these files. After going through the process, when the system detects the malicious behaviour, the YARA rules can be written, and the programable interface can update the signature.

3.1.3.2 Monitoring Process

The program, when executed, creates a process and has the associated process ID. The process can be a parent or a child process that needs to be monitored when the program is executed. For the setup, the PS utility is used to identify the usage of the active processes. Python API is calling the ps utility. The sample malware which is identified in section 3.3.3.1 is executed as part of the program. The malware was executed one at a time to observe the process activities. The process ID, CPU utilization and files execution that creates the process is being recorded. The PS utility is the interactive utility therefore iterative script is executed to send the process details at regular intervals. The interval is scheduled in such a way is to avoid the period when the malicious activity by the process is skipped during that interval session.

3.1.3.3 Maintaining Database

The information is collected in the database to maintain the state of the system processes. The interface is set up to send the process details to the database. The two components are essential for deployment. First is the database, and second is the interface that can collect the details from the system and store them in the database. Mysql is deployed to receive the data input via Python API. Only a single database is created which records the data in a tabular form, and each different field includes process ID, user details, CPU utilization, file path for the executable. The setup has been executed for a limited number of days when the test is to be conducted; therefore, database purging and achieving is not considered.

The Logs are flushed regularly with the regular internal command to keep the database running efficiently. The database role is to maintain the information inserted via API for the process details. The second API role is to read the data from the same database to populate the data for checking the abnormal behaviour, as discussed in section 3.3.3.4. Python executes the "select" statement to fetch the details. The select statement uses the connection.cursor() moreover, cursor.execute() method to retrieve the information. The setup authentication is also maintained through the local username-password database of MySQL. To avoid complexity root account is configured on the API call to access the database.

3.1.3.4 Abnormal Behaviour

The data was collected and maintained in the database to detect the abnormal behaviour in the endpoint. It gives an understanding of the standard machine process behaviour. In the setup, the data collection was not more than 2 hours. It is a controlled environment hence data collection for standard scenarios is not critical. The sample malware files were executed for both scenarios, scenario1 when known malware is executed, and signatures for the ClamAV is available. In this case, the file was blocked by the anti-virus. In the scenario2, known malicious files were executed for which

signatures were disabled. Different malware had different behaviour on the process IDs and different abnormal behaviour. The behaviour includes the detection of downloading or installing a trojan which enables the backdoor for remote access. It includes stealing the credential by downloading software that can install the key logger and transfer the keylogging to the remote user. The anomaly behaviour may include process injection, which can create a copy of the legit application and share the privilege and the memory details to the remote attacker. When it comes to analysing the behaviour of malware, the network activity is also monitored. Because it can be challenging for malicious software to significantly modify its network behaviour and still achieve its objectives, analysing malware network behaviour may provide an opportunity to identify malware on affected hosts. The process requires extracting features from network records and then creating patterns that help identify malware intrusions. The anomaly behaviour detects the files and, as per the process, send the file for the verification purpose.

3.1.3.5 File verification

The file verification is first performed by the public threat intelligence repository such as virustotal. The API calls uploads the malicious file and validate the malicious content. If the file is identified as clean, then the file is requested to virustotal for the sandbox. The next step is to update the signature of the identified malicious file. The YARA rule is created for the malicious file, but first, file analysis is done to identify the matching criteria for the file. The file analysis includes the identification of the critical pattern which are required to be matched. The pattern varies with the different types of malware files, which also depend on the artefact of possible exploits by the malware.

3.1.3.6 Action

The signature update required the ClamAV to update the database. The clamscan is executed using the API, which detects the malicious file. The action for the setup is to

kill the associated processes, including parent and child. It then deletes the files from the path identified in the ps utility output stored in the database.

3.2 Conclusion

The research design was designed to elaborate the process to identify the zero-day malware if the signature is not available on the traditional antivirus. It delineated the main research questions and explained the data source of the malware. The sample malware was taken from a public repository and signature database. It explained the requirement to remove signatures from known malware from the local antivirus tool. It has discussed the testing procedure and highlighted the research goals in the form of the hypothesis.

The results-driven in this chapter are presented in chapter 4. The output and the extracted process are also explained. The analysis results drive the process for zero-day malware detection by analysing the process id of multiple program execution and the update of the local antivirus signature using the YARA rule.

4. Results

The research questions were raised based on the literature review undertaken in chapter 2, highlighting the challenges of detecting zero-day malware files. This chapter presents the observations during the experiments conducted as per the design setup in chapter 3. The first section of the chapter provides the details of malware samples used in investigating the process for zero-day malware. The following section provides the details of the sample system process utilisation during the normal scenario. The normal scenario is referred to as the time when no sample files are executed and the lab setup machine is in an idle state. Then, the sample malware files are analysed, and the system process's behaviour with the known and unknown signatures in the antivirus tool. The section will also cover the results for the signature update on the antivirus with the integration of YARA rules. The last section presents the conclusion driven from the experiment results and answers the questions related to the research questions. The sample files used are known files to observe the method and techniques defined in chapter 3, and it does not include the detection of false positives.

4.1. Sample Files
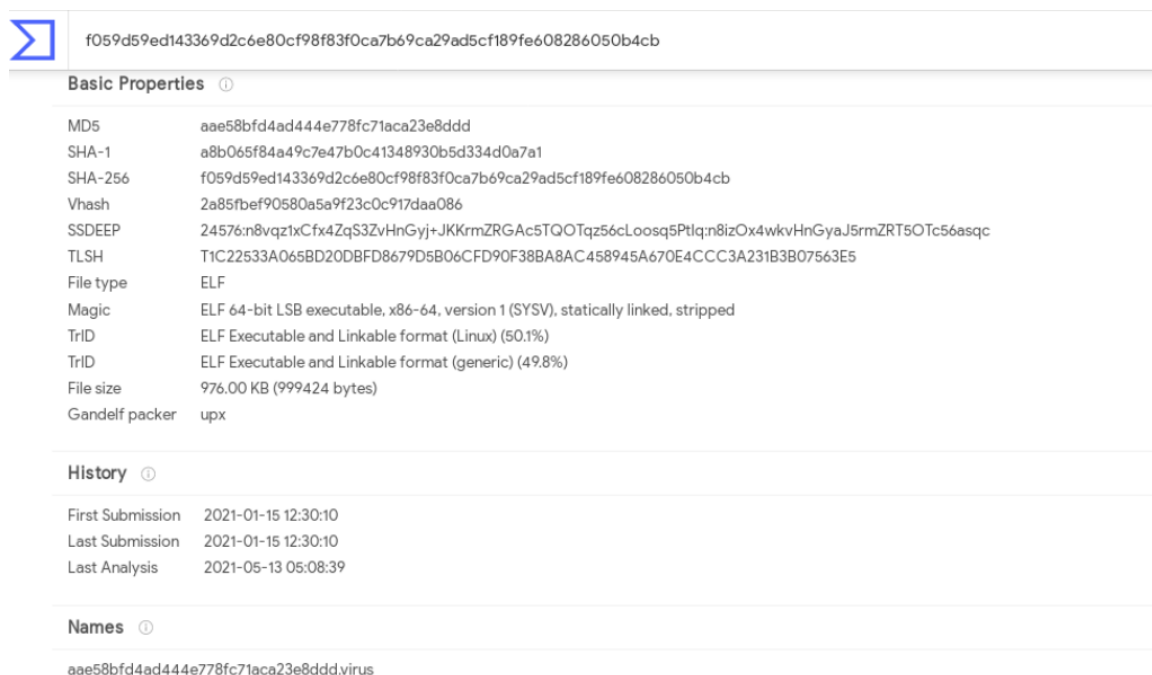
A total of seventeen (17) sample malware files were extracted from the public portal https://github.com/MalwareSamples/Linux-Malware-Samples.

Another set of five (5) non-malicious files were also used for the experiment purposes.

The following section gives an overview of the top five (5) malicious files which were verified on the virustotal to validate the signature.

Figure 4.1 shows the details of the file when it was uploaded on the virustotal. The file type is ELF (Executable and Linkable Format), which is the output of a compiler, and the

file format is binary. The file aae58bfd4ad444e778fc71aca23e8ddd.virus was first submitted to virustotal on 15/01/2021, which shows that the malware is relatively new.

Figure 4. 1- *Sample File 1 Details From VirusTotal*

Figure 4.2. shows the file details when it was uploaded on the virustotal. The second sample file type is ELF, and the file was first submitted on 01/17/2021, which is relatively new.

MD5　　7c0c01dbf6b557b4b154d84254554ff3
SHA-1　　6cd2581525bfc1f484bd67c1b38a84eb52ad8751
SHA-256　　ff4816dd923e0c7d2806c9928ed29396133cc1f81ed40a47c8e748c366811448
Vhash　　3019473411e84612b66b236153fae087
SSDEEP　　1536:817HBry6rXyJ1NL1JWP2+oGEte4ssd33OMVx8IxAmoIVuOVjeDlOXQ:8ls6rCNbWPeFzssdnOMVx8QmrVuOVyDb
TLSH　　T1B2934B23A652C67FC0C796B42BEB85219423B4390F37725A73E47DE92F169C92E5E301
File type　　ELF
Magic　　ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
Telfhash　　t12611cb0260fa89286bb219249cbc43b0169126237351beb0bf0dc584a93b002a979ecb
TrID　　ELF Executable and Linkable format (Linux) (50.1%)
TrID　　ELF Executable and Linkable format (generic) (49.8%)
File size　　87.34 KB (89435 bytes)

History ⓘ

First Submission　　2021-01-17 12:50:48
Last Submission　　2021-01-17 12:50:48
Last Analysis　　2021-01-17 12:50:48

Names ⓘ

pXdN91.x68

Figure 4. 2 - *Sample File 2 Details From VirusTotal*

Figure 4.3. shows the file details when it was uploaded on the virustotal. The third sample

file type is ELF, and the file was first submitted on 01/11/2021, which is a relatively new.

Basic Properties ⓘ

MD5　　0183b384a0920d8f9e4c04bdf03d9d48
SHA-1　　4aea68f9339d2b76cfd12a7c0550caf6d099acf0
SHA-256　　a976e428a975a145a3eb1e77c45d8a6b41ba94bd7e5192f91278026504c89e26
Vhash　　3019473411e84612b66b236153fae087
SSDEEP　　1536:27ONRd1971VlYmFbLFLA3Ux4hG3B4679l5Mh/oD+JimCJxWoGjyyZ3i5:Ya1Z1O9fFLUUxP3m/oqomaxWoGuyZ3i5
TLSH　　T1CA835C27B951C67BC0CB46F82BDFD9225C23B4B91B22620773D87DE92F128D91D59B02
File type　　ELF
Magic　　ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
Telfhash　　t13311d00331bac92d6bf668245cbc47f5155127233351beb1bf19c5849937002a975e8b
TrID　　ELF Executable and Linkable format (Linux) (50.1%)
TrID　　ELF Executable and Linkable format (generic) (49.8%)
File size　　84.83 KB (86864 bytes)

History ⓘ

First Submission　　2021-01-11 12:24:14
Last Submission　　2021-01-11 12:24:14
Last Analysis　　2021-03-01 07:35:02

Names ⓘ

bash

Figure 4. 3 - *Sample File 3 Details From VirusTotal*

Figure 4.4. shows the file details when it was uploaded on the virustotal. The fourth sample file type is ELF, and the file was first submitted on 01/12/2021, which is relatively new.

Figure 4. 4 - *Sample File 4 Details From VirusTotal*

Figure 4.5. shows the file details when it was uploaded on the virustotal. The fifth sample file type is ELF, and the file was first submitted on 01/17/2021, which is a relatively new.

Basic Properties ⓘ

MD5	16c3dd1a12cb11b56112a218c7e7d9cf
SHA-1	31d9649314b720a6b53019e1e02cc30325779c45
SHA-256	ae01f922c0918a8ad61ccedcad89326b4ebe78b7c61c54c33149f348fa9fcedb
Vhash	75b93c27f7e4b742fe34053cabcc76b0
SSDEEP	196608:mVSSXaCRjjwHXmxkPR3PhMGrYX6C9YXmsJcwZY33LWJvieDku7:mVSo4HPR3PhM5XzYnnZYnLWI
TLSH	T1B29633E1B122D202D1F09930E406FC3995F7F9256A271A4F8FC27B55D8CA791C9AB2F4
File type	ELF
Magic	ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.2.0, stripped
Telfhash	t11ee0c0808932261b23a2cd3088011b9854335617dc789e00bf7cc2d5704200b7355c7a
TrID	ELF Executable and Linkable format (Linux) (50.1%)
TrID	ELF Executable and Linkable format (generic) (49.8%)
File size	8.96 MB (9392032 bytes)

History ⓘ

First Submission	2021-01-12 06:38:20
Last Submission	2021-01-12 06:38:20
Last Analysis	2021-05-15 15:31:14

ELF Info ⓘ

Figure 4. 5 - *Sample File 5 Details From VirusTotal*

Table 4.1. summarizes the details of the remaining sample files used during the experiment to verify the process for detecting malicious files due to abnormal behaviour.

Table 4. 1 *List of Other Sample Malware Used in the Lab Setup*

File Name	File Name Used in Experiment	MD5 File Hash	File Size
Upx	Sample 6	0b9d850ad22de9ed4951984456e777897 93017e9df41271c58f45f411ef0c3d2	4.97 MB
xor2.exe	Sample 7	10995106e8810a432ebc487fafcb7e4211 00eb8ac60031e6d27c8770f6686b4e	16.84 KB
Ybf	Sample 8	1328f1c2c9fe178f13277c18847dd9adb94 74f389985e17126fcb895aac035f2	20.53 KB
Roblox	Sample 9	2023eafb964cc555ec9fc4e949db9ba3ec2 aea5c237c09db4cb71abba8dcaa97	25.62 KB
eicarstrongpassword.exe	Sample 10	2f0b2160470e2253dc6a5c9cf950962c599 9ee209d0eb0db237a4c630cb34e7a	470.69 KB
Andy	Sample 11	2f331c4e9e33c2afb8050a9a81a6775542 000e2be810104691eb9fff4981bc56	2.18 MB
Dns	Sample 12	9a7f32e59380deaedad632f1a40697d5ef4 03349f322c502599e99d81bc6dca3	512.00 KB
XMRIG	Sample 13	00ae07c9fe63b080181b8a6d59c6b3b6f9 913938858829e5a42ab90fb72edf7a	7.71 MB
real_live	Sample 14	03bb1cfd9e45844701aabc549f530d56f16 2150494b629ca19d83c1c696710d7	2.49 MB
Help	Sample 15	04b5e29283c60fcc255f8d2f289238430a1 0624e457f12f1bc866454110830a2	5.47 MB
view_list	Sample 16	0e492a3be57312e9b53ea378fa09650191 ddb4aee0eed96dfc71567863b500a8	49.25 KB
unlucky	Sample 17	1ea3dc626b9ccee026502ac8e8a98643c6 5a055829e8d8b1750b2468254c0ab1	37.90 KB
keepass	Sample 18	785565ac4cf379c857f97890070e7f82afdf 72f9e65e1a71902732af0fc00110	3.2 MB
harden.bin	Sample 19	9582fec10e6ca488ab506a96dfeb5da56c 9425ca32a8481e060bd06893fb1b3e	250 KB
Npee	Sample 20	61dc901b9962d392d781e07977852d2cbf db8bb58ca337fdc6f12081f33518dc	190 KB
abc.tar	Sample 21	d0711f6d1fd8592385999f7ade4353a695f 802fd96501df6a3e985ca195ab0d2	6.5 MB
unknown	Sample 22	8ea9fc21543e30c773adbcfac3759e8787d e315ac25e2da7f8580cb670f68424	5.5 MB

Selected Sample malware files [Sample 1 – Sample 17] used in the experiment are already identified as known malware. The ClamAV antivirus tool used in the lab setup has the signature updates available for all the selected samples except sample 5; hence, it considered all these files malicious. Signatures were removed from the antivirus signature database to treat these files as zero-day malware. The only malicious file name "sample4" signature is kept in the antivirus tool to verify if the process detects the other malicious file. Sample files from 18 to 22 are confirmed non-malicious files. Therefore, no change is required on the antivirus signature database. Files used to experiment

process defined in the research design covers zero-day malware [Sample 1-3, Sample 5-17], known malware [Sample 4] and normal file [Sample 17-22]. The purpose of the experiment does not include the detection of false-positive, and the range of samples are used to verify the effectiveness of the process.

4.2. Process Monitoring

It is essential to record the current usage of the processes in normal situations to detect the system's unusual behaviour. The process monitoring is initially required to maintain the process information before the manual execution of the malicious file. The process details are stored in the SQL database, which is then used to compare any unusual process usage due to any malicious activity. The baseline usage is set for the experiment phase. Figure 4.6 is the sample percentage of CPU usage during the usual situation with the time interval of 5 min.
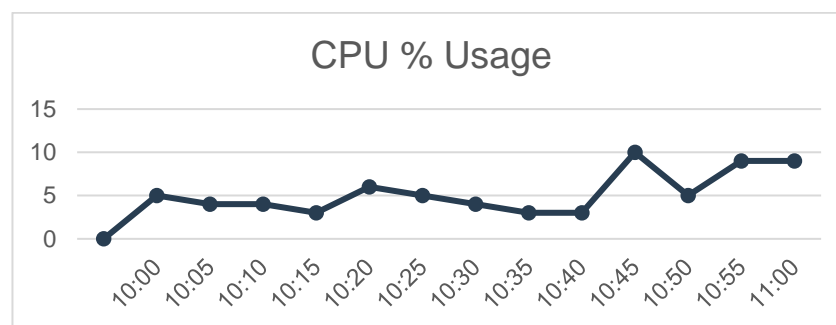
Figure 4. 6 - % CPU Usage During Normal Situation

The database maintains the process ID, CPU usage and path location for each entry. The abnormal usage of CPU utilization for the process ID triggers the alarm to the client antivirus, which marks the file as malicious. As per the test results in section 4.4, the file is deleted after the detection. The data collection is for a very limited duration and does not include any machine learning capabilities.

The process to detect abnormal behaviour requires a baseline to be set. The scope of the thesis does not cover the machine learning capability. Therefore, two components are considered for the abnormal behaviour of the system, which deviates from the baseline. These include CPU usage and the number of times the specific file is executed in a certain time frame. As the purpose of the work does not include any specific machine learning technique for abnormal behaviour; therefore, detection of false positives is also not considered.

4.3. File Analysis

File 1 – The sample file Y0hWhAUBjQ.virus was copied on the client machine and executed. During the execution, the CPU utilization went high and figure 4.7 shows the utilization during the execution. The utilization went high at approx. 10:55 when the file was executed.
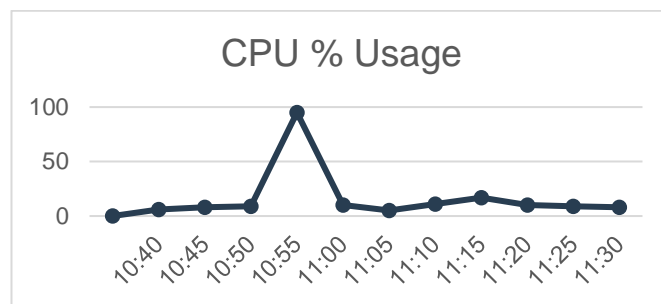
Figure 4. 7 - % CPU Usage When Sample File 1 Was Executed

Table 4.2 shows the process IDs which were active during the file execution phase.

Table 4. 2 - Sample File 1 PID

PID	Path
4586	/tmp/Y0hWhAUBjQ.virus
4599	New Fork
4599	bin/sh -e /proc/self/fd/9
4600	New Fork
4600	Date
4601	New Fork
4601	/usr/bin/python3 /usr/share/*abrt*/*abrt*-checkreports --system
4626	New Fork
4626	/bin/sh -e /proc/self/fd/9
4630	New Fork
4630	Date
4636	New Fork
4636	/usr/bin/python3 */etc/libreport*
4653	/bin/sh -e /proc/self/fd/9
4654	New Fork
4654	Date
4663	New Fork
4663	/usr/bin/python3 /usr/share/*abrt*

Figure 4.8 refers to the list of process IDs which were generated and forked by the parents process.
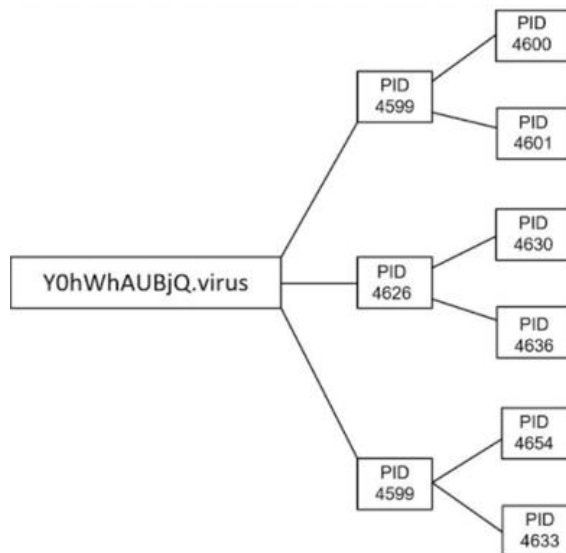
Figure 4. 8 - Sample File 1 PIDs Map

The process analysis indicates that the program is executing the directory enumeration.

The process IDs show the execution of ABRT (automatic bug detecting tool) on the user

machine. The tool helps in creating the information of the directory. The malicious file executes the python, which runs the enumeration to identify the list of directories.

Sequence of file path were detected during the time of the file execution. The file path indicates Python distributed package was accessed /usr/local/lib/python3.5/dist-packages which executed the file path /usr/bin/python3 /usr/share/abrt/abrt-checkreports –system and started running the report for /usr/bin/python3 /etc/libreport.

Table 4.3 shows the number of times the file paths were executed, which has the malicious file.

Table 4. 3 - Sample File 1 Path

File Path	Mode	Status	Count
/tmp/Y0hWhAUBjQ.virus	bits: - usr: -x grp: all: -	successful	8
/tmp/Y0hWhAUBjQ.virus	bits: - usr: -x grp: all: -	successful	10

Figure 4.9 shows the comparison of other file paths accessed while the malicious file was executed. The high count of the file execution and the high CPU utilization can be seen in Figure 4.9 and Figure 4.7.
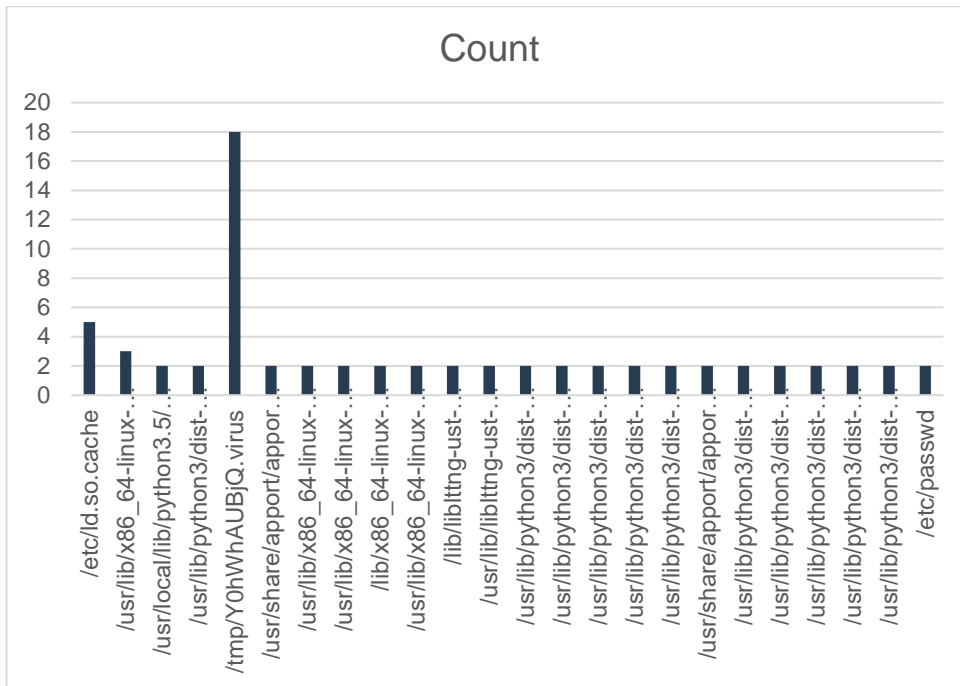
Figure 4. 9 – Sample File 1 Execution Count

After the malicious file activity is detected, the next step is to write the YARA rule for the identified file. The file location is identified in the/tmp folder. The tool used to detect the file type is the hex dump. Figure 4.10 shows the file type information. As mentioned in section 4.1 the file type of the first sample is ELF which is confirmed by the hex dump output in figure 4.9. The hex value 7f 45 4c 46 confirms the file type (Kessler, 2021).

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  d0 3e 5d 00 00 00 00 00  |..>......>].....|
00000020  40 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  03 00 40 00 00 00 00 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  eb 47 1d 00 00 00 00 00  eb 47 1d 00 00 00 00 00  |.G.......G......|
00000070  00 10 00 00 00 00 00 00  01 00 00 00 06 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  00 50 5d 00 00 00 00 00  |.........P].....|
00000090  00 50 5d 00 00 00 00 00  00 00 00 00 00 00 00 00  |.P].............|
```

Figure 4. 10 - Sample File 1 Hex dump

Figure 4.10 also has the information used from the magic number to set the content to match the YARA rule condition. The following configuration shows three variables assigned the values: 'a', 'b' and 'c'. The first variable, 'a' contains the magic number, whereas variables 'b' and 'c' are assigned other values based on the hex dump of the

remainder of the file. The condition applied that if all the values match in a file that put the description as "Sample1 virus file".

{

Meta:

desc = "Sample 1 virus file"

strings:

$a = {7f 45 4c 46 02 01 01 00}

$b= {07 78 49 4d 20 55 7a 49}

$c= {69 31 f6 fd 24 0d 2a 2e}

Condition:

$a and $b and $c

}

The signature of "sample file 1" is removed from the anti-virus signature database. Figure 4.11 shows the Clam-AV database directory listing to ensure that signature is not available before the file execution. Current files shown in the directory are not relevant and It is only to provide the comparison of the results for the automatic generation of the signature before and after the execution of the malicious file.

```
[root@ovaisahmed clamav]# ls
bytecode.cld  bytecode.cvd  daily.cld  main.cld  main.cvd  mirrors.dat
[root@ovaisahmed clamav]#
```

Figure 4. 11 - Sample File 1 AV Directory Listing Before Signature Update

The next step after the YARA rule is created to update the signature of the anti-virus tool. Figure 4.12 shows the ClamAV database after the YARA signature is updated. An API is executed written in the Python the to normalize the YARA Rule file automatically and then uses the command clamscan --leave-temps --tempdir=mytempdir (Alvarez, 2021). The file name Y0hwhAUBjQ.yara is created under the anti-virus signature database.

```
[root@ovaisahmed clamav]# ls
bytecode.cld   daily.cld   main.cvd      Y0hWhAUBjQ.yara
bytecode.cvd   main.cld    mirrors.dat
[root@ovaisahmed clamav]#
```

Figure 4. 12 - *Sample File 1 AV Directory Listing After Signature update*

The next step is to remove the malicious file from the identified location. API is executed to run the scan. It is executed on the folder's location, which in this case is at /tmp location. The anti-virus tool has the signature written as a YARA rule to detect the file as malicious and remove it from the location. Figure 4.12 shows the scan output, which indicates that one infected file has been deleted. The end time is the time when the scan was completed and the malicious file was deleted. The size of the file indicated in figure 4.13 is the same as the file of the malicious file.

```
---------- SCAN SUMMARY ----------
Known viruses: 8536763
Engine version: 0.103.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 999.4 kB
Data read: 999.4 kB(ratio 0.00:1)
Time: 15.334 sec (0 m 15 s)
Start Date: 2021:05:25 10:57:47
End Date:   2021:05:25 10:58:03
```

Figure 4. 13 - *Sample File 1 AV Scan Result*

In summary, the malicious sample file one was executed while the process usage, file and CPU utilization behaviour were monitored. The database contains the information of process ID, CPU utilization, and the file's path executed before and after the malicious file execution. The abnormal behaviour of the processor utilization of the file marked the file as malicious. The identification allowed to creation of a YARA rule specific to the file. The magic number of the file is taken for file identification using the hexdump utility. The YARA rule signature updated the antivirus signature database. In the end, an antivirus

scan was executed for the identified malicious path to delete the malicious file. The file was removed, and the process monitoring was put back to the normal situation.

File 2: The sample file pXdN91.x68 was copied on the client machine and executed. During the executions, the CPU utilization went high and figure 4.14 shows the utilization during the execution. The utilization went high at approx. 17:55 when the file was executed.
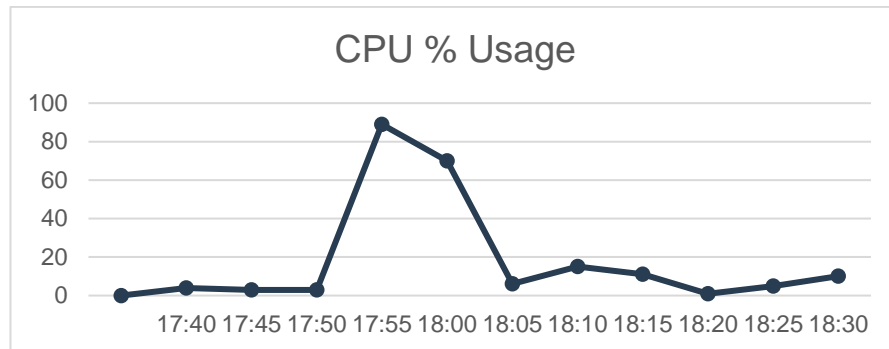
Figure 4. 14 - % CPU Usage When Sample File 2 Was Executed

Table 4.4 shows the process IDs which were active during the file execution phase.

Table 4. 4 - Sample File 2 PID

PID	Path
4582	/tmp/pXdN91.x68
4597	New Fork
4598	New Fork

Figure 4.15 refers to the list of process IDs that were generated and forked by the parent's process.
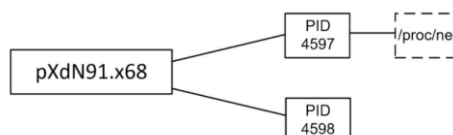
Figure 4. 15 - Sample File 2 PIDs Map

The process analysis indicates that the malicious software is trying to get the remote discovery of the system. The remote discovery enables the lateral movement such that malicious actors can hop from the compromised machine to the other machine in the network. The use of the/net/route process identifies that the Remote discovery technique is used (Stepanic, 2017).

The sequence of file paths was detected during the time of the file execution. The file path indicates that multiple processes for route /proc/4582/net/route.

Table 4.5 shows the number of times the file paths were executed, which has the malicious file.

Table 4. 5 - Sample File 2 Path

File Path	Mode	Status	Count
/tmp/pXdN91.x68	bits: - usr: - grp: - all: -	successful	1
/tmp/pXdN91.x68	bits: - usr: - grp: - all: -	successful	256

Figure 4.16 shows the comparison of other file paths accessed while the malicious file was executed. The other file was executed only once. That is why not visible in figure 4.16. The high count of the file execution and the high CPU utilization can be verified in Figure 4.16 and Figure 4.14.
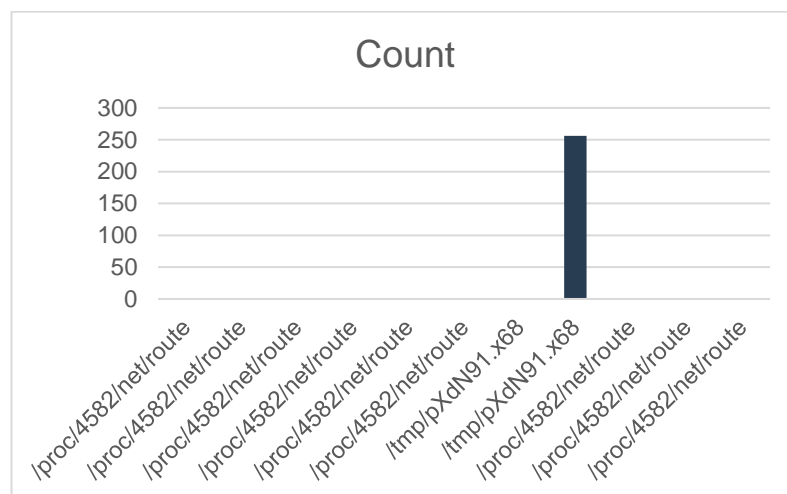
Figure 4. 16 – Sample File 2 Execution Count

After the malicious file activity is detected, the next step is to write the YARA rule for the identified file. The file location is identified in the/tmp folder. The tool used to detect the file type is the hex dump. Figure 4.17 shows the file type information. As mentioned in section 4.1, the file type of the second sample file is ELF which is confirmed by the hex dump output in figure 4.17. The hex value 7f 45 4c 46 confirms the file type (Kessler, 2021).

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  94 01 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  a8 f9 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  03 00 40 00 0f 00 0c 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  24 e9 00 00 00 00 00 00  24 e9 00 00 00 00 00 00  |$.......$.......|
00000070  00 00 10 00 00 00 00 00  01 00 00 00 06 00 00 00  |................|
00000080  28 e9 00 00 00 00 00 00  28 e9 50 00 00 00 00 00  |(.......(.P.....|
00000090  28 e9 50 00 00 00 00 00  d0 05 00 00 00 00 00 00  |(.P.............|
```

Figure 4. 17 - Sample File 2 Hex dump

Figure 4.17 also has the information used from the magic number to set the content to match the YARA rule condition. The following configuration shows three variables assigned the values: 'a', 'b' and 'c'. The first variable, 'a' contains the magic number, whereas variables 'b' and 'c' are assigned other values based on the hex dump of the remainder of the file. The condition applied if all the values match in a file that put the description as "Sample 2 virus file".

Rule sample2

{

Meta:

desc = "Sample 2 virus file"

strings:

$a ={7f 45 4c 46 02 01 01 00}

$b={5f 5f 47 49 5f 5f 5f 65 }

$c={74 73 69 64 00 5f 5f 47}

Condition:

$a and $b and $c

}

The signature of "sample file 2" is removed from the anti-virus signature database. Figure 4.18 shows the Clam-AV database directory listing to ensure that signature is not available before the file execution. Current files shown in the directory are not relevant and It is only to provide the comparison of the results for the automatic generation of the signature before and after the execution of the malicious file.

.


```
[root@ovaisahmed clamav]# ls
bytecode.cld  daily.cld  main.cvd      Y0hWhAUBjQ.yara
bytecode.cvd  main.cld   mirrors.dat
[root@ovaisahmed clamav]#
```

Figure 4. 18 - *Sample File 2 AV Directory Listing Before Signature Update*

The next step after the YARA rule is to update the signature of the anti-virus tool. Figure 4.19 shows the ClamAV database after the YARA signature is updated. The API calls the YARA rule file to normalize it automatically and then uses the command clamscan --leave-temps --tempdir=mytempdir (Alvarez, 2021). The file name pXdN91.yara is created under the anti-virus signature database.


```
[root@ovaisahmed clamav]#
[root@ovaisahmed clamav]# ls
bytecode.cld  daily.cld  main.cvd      pXdN91.yara
bytecode.cvd  main.cld   mirrors.dat  Y0hWhAUBjQ.yara
[root@ovaisahmed clamav]#
```

Figure 4. 19 - *Sample File 2 AV Directory Listing After Signature update*

The next step is to remove the malicious file from the identified location. API is executed to execute the scan. It is executed on the folder's location, which in this case is at /tmp location. The anti-virus tool has the signature written as a YARA rule to detect the file as malicious and remove it from the location. Figure 4.20 shows the scan output, which indicates that one infected file has been deleted. The end time is the time when the scan was completed and the malicious file was deleted. The file size indicated in figure 4.20 is the same as the file of the malicious file.

```
---------- SCAN SUMMARY ----------
Known viruses: 8536765
Engine version: 0.103.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 89.4 kB
Data read: 89.4 kB(ratio 0.00:1)
Time: 10.154 sec (0 m 10 s)
Start Date: 2021:05:25 17:57:47
End Date:   2021:05:25 17:57:57
```

Figure 4. 20 - Sample File 2 AV Scan Result

In summary, malicious "sample file two" was executed while the process usage, file and CPU utilization behaviour were monitored. The database contains the information of process ID, CPU utilization, and the file's path executed before and after the malicious file execution. The abnormal behaviour of the processor utilization of the file marked the file as malicious. The identification allowed to creation of a YARA rule specific to the file. The magic number of the file is taken for file identification using the hex dump utility. The YARA rule signature updated the antivirus signature database. In the end, an antivirus scan was executed for the identified malicious path to delete the malicious file.

File 3: The sample file bash was copied on the client machine and executed. During the execution, the CPU utilization went high and figure 4.21 shows the utilization during the execution. The utilization went high at approx. 20:05 when the file was executed.
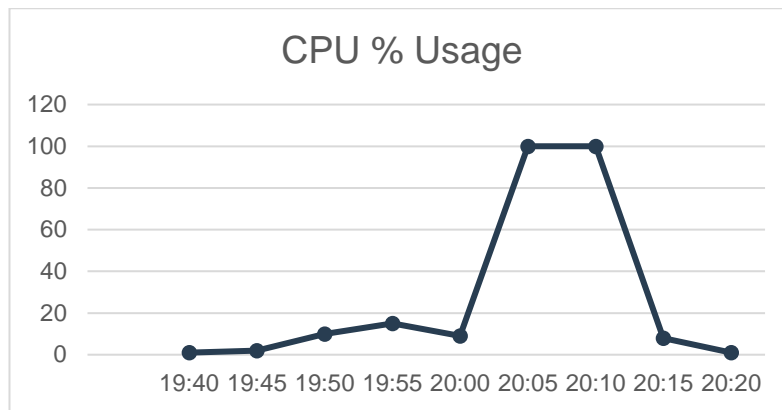
Figure 4. 21 - % CPU Usage When Sample File 3 Was Executed

Table 4.6 shows the process IDs which were active during the file execution phase.

Table 4. 6 - Sample File 3 PID

PID	Path
4582	/tmp/bash
4596	New Fork
4597	New Fork

Figure 4.22 refers to the list of process IDs which were generated and forked by the parents process.
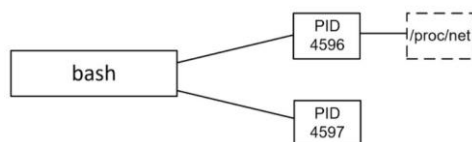
Figure 4. 22 - Sample File 3 PIDs Map

The process analysis indicates that the malicious software is trying to get the remote discovery of the system and the masquerading and non-standard port for command and control tactics of an attack. The remote discovery enables the lateral movement such that malicious actors can hop from the compromised machine to the other machine in the network. The malicious software sends the traffic to the public IP address on a non-standard port. From the details collected from process IDs, the type of attack detected in the remote discovery.

The sequence of file paths was detected during the time of the file execution. The file path indicates that multiple processes for route /proc/net/route.

Table 4.7 shows the number of times the file paths were executed, which has the malicious file.

Table 4. 7 - Sample File 3 Path

File Path	Mode	Status	Count
/tmp/bash	bits: - usr: - grp: - all: -	successful	256

Figure 4.23 shows the comparison of other file paths accessed while the malicious file was executed. The high count of the file execution and the high CPU utilization can be compared in Figure 4.23 and Figure 4.21.
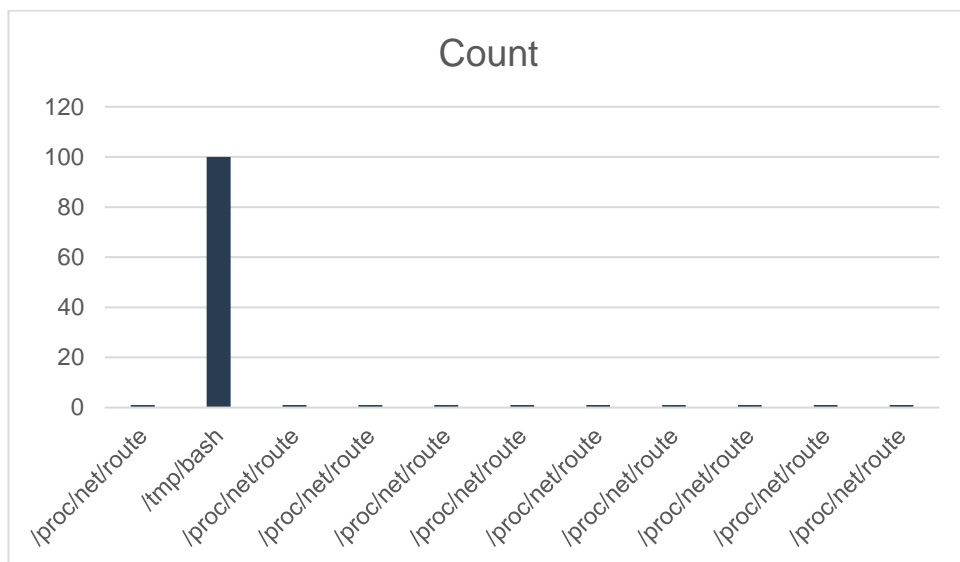
Figure 4. 23 – Sample File 3 Execution Count

After the malicious file activity is detected, the next step is to write the YARA rule for the identified file. The file location is identified in the/tmp folder. The tool used to detect the file type is the hex dump. Figure 4.24 shows the file type information. As mentioned in section 4.1, the file type of the third sample file is ELF which is confirmed by the hex dump output in figure 4.17. The hex value 7f 45 4c 46 confirms the file type (Kessler, 2021).

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  94 01 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  08 ed 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  03 00 40 00 0f 00 0c 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  54 db 00 00 00 00 00 00  54 db 00 00 00 00 00 00  |T.......T.......|
00000070  00 00 10 00 00 00 00 00  01 00 00 00 06 00 00 00  |................|
00000080  58 db 00 00 00 00 00 00  58 db 50 00 00 00 00 00  |X.......X.P.....|
00000090  58 db 50 00 00 00 00 00  18 07 00 00 00 00 00 00  |X.P.............|
```

Figure 4. 24 - *Sample File 3 Hex dump*

Figure 4.24 also has the information used from the magic number to set the content to match the YARA rule condition. The following configuration shows three variables assigned the values: 'a', 'b' and 'c'. The first variable, 'a' is assigned the file type hex value, whereas variables 'b' and 'c' are assigned other random values from the magic number. The condition applied that if all the values match in a file that put the description as "Sample 3 virus file".

Rule sample3

{

Meta:

desc = "Sample 3 virus file"

strings:

$a = {7f 45 4c 46 02 01 01 00}

$b= {6e 65 74 5f 60 64 64 72}

$c= {6e 61 6d 65 00 63 6c 6f}

Condition:

$a and $b and $c

}

The signature of "sample file 3" is removed from the anti-virus signature database. Figure 4.25 shows the Clam-AV database location to ensure that signature is not available before the file execution. It is to provide the comparison of the results for the automatic generation of the signature before and after the execution of the malicious file.

.

```
[root@ovaisahmed clamav]# ls
bytecode.cld  daily.cld  main.cvd     pXdN91.yara
bytecode.cvd  main.cld   mirrors.dat  Y0hWhAUBjQ.yara
[root@ovaisahmed clamav]#
[root@ovaisahmed clamav]#
```

Figure 4. 25 - *Sample File 3 AV Directory Listing Before Signature Update*

The next step after the YARA rule is created to update the signature of the anti-virus tool. Figure 4.26 shows the ClamAV database after the YARA signature is updated. The API calls the YARA rule file to normalize it automatically and then uses the command clamscan --leave-temps --tempdir=mytempdir (Alvarez, 2021). The file name bash. Yara is created under the anti-virus signature database.

```
[root@ovaisahmed clamav]#
[root@ovaisahmed clamav]# ls
bash.yara      bytecode.cvd  main.cld  mirrors.dat  Y0hWhAUBjQ.yara
bytecode.cld  daily.cld     main.cvd  pXdN91.yara
[root@ovaisahmed clamav]#
```

Figure 4. 26 - *Sample File 3 AV Directory Listing After Signature update*

The next step is to remove the malicious file from the identified location. API is executed to execute the scan. It is executed on the folder's location, which in this case is at /tmp location. The anti-virus tool has the signature written as a YARA rule to detect the file as malicious and remove it from the location. Figure 4.27 shows the scan output, which indicates that one infected file has been deleted. The end time is the time when the scan was completed and the malicious file was deleted. The file size indicated in figure 4.27 is the same as the file of the malicious file.

```
----------- SCAN SUMMARY -----------
Known viruses: 8536768
Engine version: 0.103.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 86.9 kB
Data read: 86.9 kB(ratio 0.00:1)
Time: 9.018 sec (0 m 9 s)
Start Date: 2021:05:27 18:00:00
End Date:   2021:05:27 18:00:09
```

Figure 4. 27 - *Sample File 3 AV Scan Result*

In summary, malicious sample file three was executed while the process usage, file and CPU utilization behaviour were monitored. The database contains the information of process ID, CPU utilization, and the file's path executed before and after the malicious file execution. The abnormal behaviour of the processor utilization of the file marked the file as malicious. The identification allowed to creation of a YARA rule specific to the file. The magic number of the file is taken for file identification using the hex dump utility. The YARA rule signature updated the antivirus signature database. In the end, an antivirus scan was executed for the identified malicious path to delete the malicious file.

File 4: The sample file x86 was copied on the client machine and executed. During the execution, the CPU utilization went high and figure 4.28 shows the utilization during the execution. The utilization went high at approx. 20:05 when the file was executed.
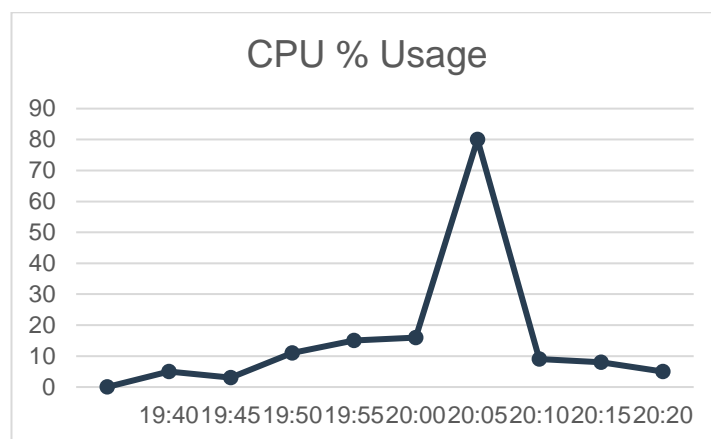
Figure 4. 28 - % CPU Usage When Sample File 4 Was Executed

Table 4.8 shows the process IDs which were active during the file execution phase.

Table 4. 8 - Sample File 4 PID

PID	Path
4583	/tmp/x86
4592	New Fork
4593	New Fork
4594	New Fork
4597	New Fork

4598	New Fork
4595	New Fork
4596	New Fork

Figure 4.29 refers to the list of process IDs which were generated and forked by the parents process.
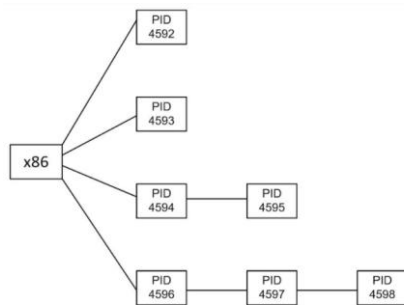
Figure 4. 29 - *Sample File 4 PIDs Map*

The process analysis indicates that the malicious software enabled multiple processes, which led to masquerading, operating system credential dumping, remote discovery, and the non-standard ports and application layers. The process ID could not track the non-standard port and the credential dumping for the operating system. The /proc/net/route indicates the use of remote discovery, and the execution of /dev/watchdog indicates that the code execution was to know about the system details. The /proc/410/maps lead to masquerading the ports mapping at the application layer.

Table 4. 9 - *Sample File 4 Path*

File Path	Mode	Status	Count
/tmp/x86	bits: - usr: - grp: - all: -	successful	10

Figure 4.30 shows the comparison of other file paths accessed while the malicious file was executed. The high count of the file execution and the high CPU utilization can be compared in Figure 4.30 and Figure 4.28.
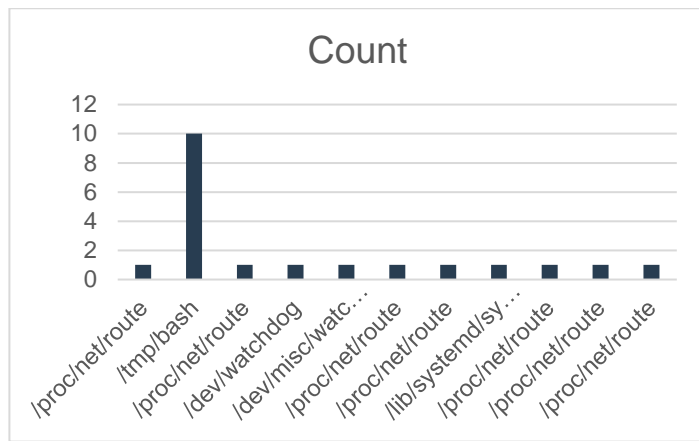
Figure 4. 30 – Sample File 4 Execution Count

After the malicious file activity is detected, the next step is to write the YARA rule for the identified file. The file location is identified in the/tmp folder. The tool used to detect the file type is the hex dump. Figure 4.31 shows the file type information. As mentioned in section 4.1, the file type of the second sample file is ELF which is confirmed by the hex dump output in figure 4.17. The hex value 7f 45 4c 46 confirms the file type (Kessler, 2021).

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  94 01 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  50 96 01 00 00 00 00 00  |@.......P.......|
00000030  00 00 00 00 40 00 38 00  03 00 40 00 0f 00 0c 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  38 81 01 00 00 00 00 00  38 81 01 00 00 00 00 00  |8.......8.......|
00000070  00 00 10 00 00 00 00 00  01 00 00 00 06 00 00 00  |................|
00000080  38 81 01 00 00 00 00 00  38 81 51 00 00 00 00 00  |8.......8.Q.....|
00000090  38 81 51 00 00 00 00 00  f0 08 00 00 00 00 00 00  |8.Q.............|
```
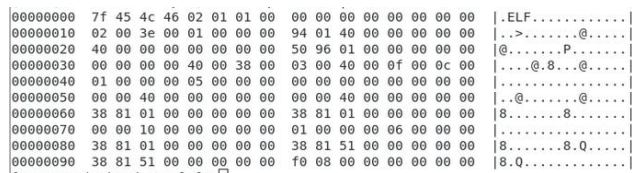
Figure 4. 31 - Sample File 4 Hex dump

Figure 4.31 also has the information used from the magic number to set the content to match the YARA rule condition. The following configuration shows three variables assigned the values: 'a', 'b' and 'c'. The first variable, 'a' is assigned the file type hex value, whereas variables 'b' and 'c' are assigned other random values from the magic number. The condition applied that if all the values match in a file that put the description as "Sample 4 virus file".

Rule sample4

{

Meta:

desc = "Sample 4 virus file"

strings:

$a = {7f 45 4c 46 02 01 01 00}

$b= {6c 30 30 64 00 66 63 6e}

$c= {70 65 5f 74 6f 6c 6f 77}

Condition:

$a and $b and $c

}

The signature of "sample file 4" is removed from the anti-virus signature database. Figure 4.32 shows the Clam-AV database location to ensure that signature is not available before the file execution. This is to provide the comparison of the results for the automatic generation of the signature before and after the execution of the malicious file.

```
[root@ovaisahmed clamav]#
[root@ovaisahmed clamav]# ls
bash.yara      bytecode.cvd  main.cld  mirrors.dat  Y0hWhAUBjQ.yara
bytecode.cld  daily.cld      main.cvd  pXdN91.yara
[root@ovaisahmed clamav]#
```

Figure 4. 32 - Sample File 4 AV Directory Listing Before Signature Update

The next step after the YARA rule is created to update the signature of the anti-virus tool. Figure 4.12 shows the ClamAV database after the YARA signature is updated. The API calls the YARA rule file to normalize it automatically and then uses the command clamscan --leave-temps --tempdir=mytempdir (Alvarez, 2021). The file name x66.yara is created under the anti-virus signature database.

```
[root@ovaisahmed clamav]# ls
bash.yara      bytecode.cvd  main.cld  mirrors.dat  x66.yara
bytecode.cld  daily.cld      main.cvd  pXdN91.yara  Y0hWhAUBjQ.yara
[root@ovaisahmed clamav]#
```

Figure 4. 33 - Sample File 4 AV Directory Listing After Signature update

The next step is to remove the malicious file from the identified location. API is executed to execute the scan. It is executed on the folder's location, which in this case is at /tmp location. The anti-virus tool has the signature written as a YARA rule to detect the file as malicious and remove it from the location. Figure 4.34 shows the scan output, which indicates that one infected file has been deleted. The end time is the time when the scan was completed, and the malicious file was deleted. The file size indicated in figure 4.34 is the same as the file of the malicious file..

```
----------- SCAN SUMMARY -----------
Known viruses: 8536768
Engine version: 0.103.2
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 134.2 kB
Data read: 134.2 kB(ratio 0.00:1)
Time: 18.010 sec (0 m 18 s)
Start Date: 2021:05:27 20:05:01
End Date:   2021:05:27 20:05:19
[root@ovaisahmed log]
```

Figure 4. 34 - Sample File 4 AV Scan Result

In summary, malicious sample file two was executed while the process usage, file and CPU utilization behaviour were monitored. The database contains the information of process ID, CPU utilization, and the file's path executed before and after the malicious file execution. The abnormal behaviour of the processor utilization of the file marked the file as malicious. The identification allowed to creation of a YARA rule specific to the file. The magic number of the file is taken for file identification using the hexdump utility. The YARA rule signature updated the antivirus signature database. In the end, an antivirus scan was executed for the identified malicious path to delete the malicious file.

File 5: The file execution of the sample4 creates the following process IDs.

Table 4. 10 - Sample File 5 PID

PID	Path
4567	/tmp/sample4
4576	New Fork
4576	/tmp/sample4

The signature of the files was kept on the anti-virus signature database. Therefore, as soon as the file was executed, the file was detected and deleted. The file execution of the sample file was run in a controlled way in such a way that the anti-virus scan was not executed before the file run. Hence during the run phase, the anti-virus detects the malicious behaviour and deletes the file.

Other Sample Files:

Figure 4.35 depicts the relationship between the CPU usage and the number of times the file is executed. The details of the malicious files is shown in Table 4.1.
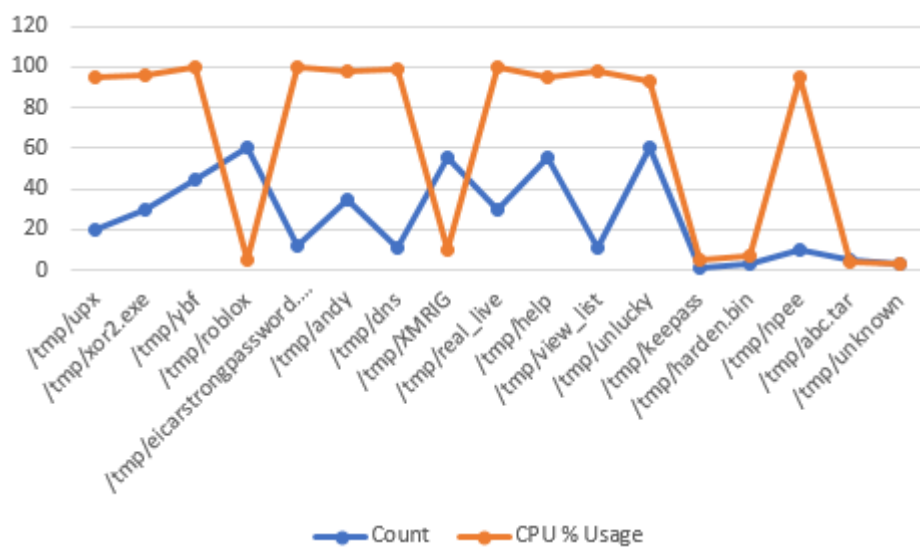
Figure 4. 35 - Other Sample Files % CPU Usage and Execution Count

Multiple files were executed to observe the results to experiment with the automated process as defined in chapter 3. Sample files from 18 to 22 are known non-malicious files. Therefore, no changes were required to update the signature of the endpoint anti-virus tool. These results also underwent the same automatic process where the file execution detects the abnormal activity, identifies the file, creates the YARA rule to update the antivirus tool. And in the end, the automated API call runs the scan to quarantine the file.

The experiment results above help in answering the following research question.

Question 1 (Q1) – Which system process on Linux system can detect anomaly behaviour?

Different sample files on the Linux system were executed in different time intervals. Two categories of sample files were executed. The first category of sample files was known malicious; however, signatures were removed from the antivirus tool. Each malicious file was executed to analyse the behaviour of the system. It was observed that most of the files in the first category were detected with high CPU utilisation and the number of times files were executed. It followed the same detection criteria as it is defined in section 4.3. The abnormal behaviour marked the file as malicious. The next set of sample files was normal, and most of the files were undetected as the CPU utilisation did not change drastically, nor was the file execution count was high. Discussion on the experiment results is explained in chapter 5 which also provides the briefing of the unsuccessful results.

Question 2 (Q2)- How can YARA rules and antivirus software be integrated after the zero-day malware is detected?

YARA rules and antivirus tool were integrated to act, so if the file was detected due to anomaly behaviour could be actioned as the zero-day malicious file. The process flow in figure 3.3 indicates that the YARA rule is written automatically using Python script. The signature for the antivirus tool is then updated using the YARA rule API integration. The last step is to automate the execution of the on-demand antivirus scan, which detects and delete the malicious file.

4.5. Conclusion

The result has reported multiple processes IDs and the sample file paths executed during the experiment phase. The automation process involving API calls at different stages

was also presented with the test output. The research questions were also answered with the results. The next chapters will focus on the dataset collected during the experiment phase and the method's effectiveness to use anomaly process detection to find the zero-day malware.

5. Discussion

In chapter 4, the output of the experiment results was presented driven from the setup as briefed in chapter 3. This chapter provides the analysis of results collected in chapter 4 and discusses the challenges of zero-day malware detection as outlined in chapter 2. The findings in chapter 4 are used to answer the research questions as raised in chapter 3.

This chapter is divided into two main sections, section 5.2 discusses the sample data set, and section 5.3 discusses the output of the test results and analyses if the output is advanced or contradict as presented in chapter 2. It answers the main research questions hence justifying the need for the study.

5.1. Sample Data Sets

The sample malware used in the lab setup is for Linux-based machines' x86 and 64 bits CPU architecture. The results depend on the types of malware used in chapter 4, as it has relied on detecting abnormal behaviour. A range of malware was considered during the experiment. Some sample malware were known malware. However, the signatures of these samples were removed from the antivirus tool due to the lab setup. The other set of samples were known non-malware to see the effect of the process as defined in chapter 3. Table 5.1 indicates the score rating for this malware across different threat intelligence tools which are publicly available. The list indicates the last five sample files are normal as these files were found to be clean on the threat intelligence websites.

Table 5. 1 - Sample Malware Rating

Sample Malware	JoeSandbox	VirusTotal	Metadefender	Reversing Lab
Y0hWhAUBjQ.virus	48/100	24/63	7/37	10/29
pXdN91.x68	56/100	31/63	17/37	16/29
bash	64/100	40/63	18/37	20/29
x86	100/100	34/63	17/37	17/29
upx	76/100	61/63	19/37	22/29
xor2.exe	88/100	54/63	27/37	29/29
ybf	100/100	45/63	28/37	15/29
roblox	78/100	31/63	14/37	18/29
eicarstrongpassword.exe	96/100	33/63	35/37	17/29
andy	86/100	59/63	16/37	21/29
dns	72/100	63/63	18/37	23/29
XMRIG	84/100	57/63	19/37	29/29
real_live	40/100	45/63	36/37	21/29
help	90/100	61/63	14/37	25/29
view_list	76/100	42/63	29/37	29/29
unlucky	68/100	39/63	25/37	13/29
keepass	no record	no record	no record	no record
harden.bin	no record	no record	no record	no record
npee	no record	no record	no record	no record
abc.tar	no record	no record	no record	no record
unknown	no record	no record	no record	no record

The complex behaviour of the malicious sample files had to be verified with the security framework mitre att&ck. The following section discusses the behaviour of the malicious activity detected by the threat intelligence. The initial discussion explains sample file behaviour from third-party malware analysis tools and not the output collected from chapter 4. The information about the known malware is essential to set the baseline for the behaviour. The action ensures that the malicious activity detected by the process defined in chapter 3 detects this known malware as malicious files. This security framework is a publicly available database based on the adversaries of tactics and techniques based on real-world examples. The knowledge base is used as a foundation for developing specific threat models and methodologies in different security

organisations, including vendors who develop the security products (Stepanic, 2017). Table 5.2 shows the reference of the sample malware categorisation.

Table 5. 2 - *Sample Categorization*

Sample Malware	Defense Evasion	Credential Access	Discovery	Command and Control	Lateral Movement	Exfilteration
Y0hWhAUBjQ.virus			Security Software Discovery			
pXdN91.x68			Remote System Discovery1	Non-Standard Port1		
bash	Masquerading		Remote System Discovery1	Non-Standard Port1		
x86	Masquerading	OS Credential Dumping1		Non-Standard Port11		
				Non-Application Layer Protocol1		
upx						Data Transfer Size Limits
xor2.exe					Internal Spearphishing	
ybf				Protocol Impersonation		
roblox			File and Directory Discovery		RDP Hijacking	
eicarstrongpassword.exe	Masquerading				Internal Spearphishing	
andy				Data Obfuscation		
dns				DNS		
XMRIG						Exfiltration Over Web Service
real_live					Internal Spearphishing	
help	Bypass User Account Control					
view_list		Password Cracking				
unlucky	Masquerading					Data Transfer Size Limits
keepass						
harden.bin						
npee						
abc.tar						
unknown						

Masquerading may attempt to manipulate their artifact features to make them appear legitimate to users and security tools. Masquerading occurs when the name or location of an object, legitimate or malicious, is manipulated or abused for the sake of evading

defences and observation. It may include manipulating file metadata, tricking users into misidentifying the file type, and giving legitimate task or service names.

OS Credential Dumping Adversary may attempt to dump credentials to obtain account login and credential material, generally in the form of a hash or a clear text password, from the operating system and software. Credentials can then be used to perform the lateral movement, and access restricted information.

Security software adversary may attempt to get a listing of software and software versions installed on a system or in a cloud environment. It may use the information from Software discovery during automated discovery to shape follow-on behaviours, including whether the adversary thoroughly infects the target and attempts specific actions. Adversaries may attempt to enumerate software for various reasons, such as figuring out what security measures are present or if the compromised system has a software version that is vulnerable to exploitation for privilege escalation.

Non-application adversaries may use a non-application layer protocol to communicate between host and server or among infected hosts within a network. Examples include the use of network-layer protocols, such as the Internet Control Message Protocol (ICMP), transport layer protocols, such as the User Datagram Protocol (UDP), session layer protocols, such as Socket Secure (SOCKS).

The behaviour of the sample file is essential to understand the technique explained in chapter 3. It is to confirm the behaviour of the file and the test conducted by the file in chapter 4 to support the same result. Sample file 5, "sample4", is used to validate the method explained in chapter 3. It is achieved by keeping the signature of the sample in the antivirus to allow it to quarantine the file immediately as it is placed on the end-point.

5.2. Research Questions

The main objective of this study was to get the answers for the following questions:

Question 1 (Q1) – Which system process on Linux system can detect anomaly behaviour?

Answer:

Multiple processes which create malicious activity on the system can detect anomaly behaviour.

Discussion:

Sample malware was taken, which contains the binaries which targeted the CPU utilization of the client machine. The CPU utilization was monitored before and after different sample files were executed during different intervals. Figure 5.1. shows the comparison of the CPU utilization compared when different sample files were executed.

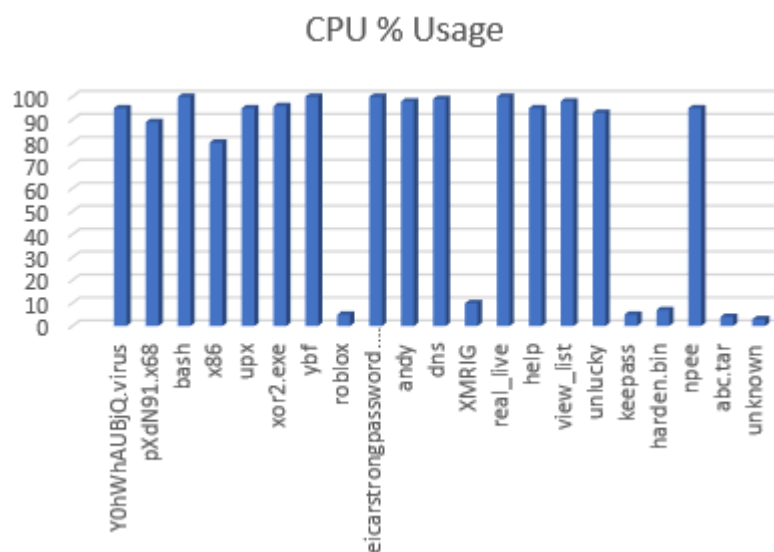*Figure 5. 1 - Compare % CPU Utilization*

As addressed in the literature review, the traditional antivirus uses the signatures-based system to detect the malware on the end-points (Mohanta et al., 2020). In a traditional environment, the malicious software analysis must be completed using static or dynamic techniques (Kara, 2019). The results in that setup lead to the creation of the signatures

populated on the threat intelligence system. Different vendors utilise feeds to update the database of the antivirus client installed on the end-point (Mohanta et al., 2020) from the threat intelligence. Chapter 2 elaborates the challenge faced by the delay caused in creating the signature database of the malicious file as it must go through the entire analysis technique. A new malicious file, which does not have the signature, remains undetected as the antivirus does not have the mechanism to detect the file and treat it as a threat (Radhakrishnan et al., 2019). It is the risk of zero-day malware, which is new and do not have the artefact. Chapter 2 elaborates on other techniques as well for the detection of zero-day malware. The challenges with the techniques require the sandboxing technique and analysis of the behaviour. Chapter 3 details the method that does not require a sandbox environment and relies on detecting the malicious activity by tracking the process behaviour concerning the CPU utilisation, active processes, and the number of times the malicious file was executed. The detection of abnormal behaviour is based on specific criteria. For the experiment purpose, the CPU usage and the execution of a specific count of a particular process would mark the file as malicious. The findings of chapter 3 show that if the setup flow is used to implement the track of the malicious activity, it will answer question 1 as raised as part of the thesis. When the sample file was executed Y0hWhAUBjQ.virus, the utilisation of the CPU went high, Figure 4.7 for reference, during the window, the malicious file was observed to be executed multiple times, and the series of the processes were detected. Table 5.3. shows the relationship of the process IDs, the file path, and the count section show the number of times the process and the file path were reported during the execution of the malicious file.

Table 5. 3 - Sample 1 Process ID and the Execution Count

PID	Path	Count
4586	/tmp/Y0hWhAUBjQ.virus	18
4599	New Fork	18
4599	bin/sh -e /proc/self/fd/9	18

4600	New Fork	18
4600	date	18
4601	New Fork	18
4601	/usr/bin/python3 /usr/share/*abrt*/*abrt*-checkreports --system	18
4626	New Fork	18
4626	/bin/sh -e /proc/self/fd/9	18
4630	New Fork	18
4630	date	18
4636	New Fork	18
4636	/usr/bin/python3 */etc/libreport*	18
4653	/bin/sh -e /proc/self/fd/9	18
4654	New Fork	18
4654	date	18
4663	New Fork	18
4663	/usr/bin/python3 /usr/share/*abrt*	18

The test result shows that process IDs are essential, but the way the process is executed makes the file malicious. The process analysis indicates that the program is executing the directory enumeration. The process ID 4601 show the execution of ABRT (automatic bug detecting tool) on the user machine. The tool helps in creating the information of the directory. The malicious file executes the Python, which runs the enumeration to identify the list of directories. The process ID 4636 indicates execution of the report collection of the data gathered by the execution of the ABRT also indicates that information is being gathered. The process ID associated with this action can be executed in the normal situation for locating the file by a regular program and running a report at a specific time using the date function; however, the combination of the malicious behaviour, which is comprised of the CPU utilization and the number of times the execution of the same program makes it malicious. The abnormal behaviour detected due to the process execution has identified the malicious file execution. Table 5.4 shows the summarized results for the other sample files used in chapter 4.

Table 5. 4 - Process ID and the Execution Count

Sample File 2 pXdN91.x68		
PID	**Path**	**Count**
4582	/tmp/pXdN91.x68	257
4597	New Fork	257

4598	New Fork	257
Sample File 3 bash		
4582	/tmp/bash	256
4596	New Fork	256
4597	New Fork	256
Sample File 4 x86		
4583	/tmp/x86	10
4592	New Fork	10
4593	New Fork	10
4594	New Fork	10
4597	New Fork	10
4598	New Fork	10
4595	New Fork	10
4596	New Fork	10
Sample File 6 upx		
4569	/tmp/upx	20
Sample File 7 xor2.exe		
4588	/tmp/xor2.exe	30
Sample File 8 ybf		
4859	/tmp/ybf	45
Sample File 9 roblox		
4587	/tmp/roblox	60
Sample File 10 eicarstrongpassword.exe		
4592	/tmp/eicarstrongpassword.exe	12
Sample File 11 andy		
4855	/tmp/andy	35
Sample File 12 dns		
4798	/tmp/dns	11
Sample File 13 XMRIG		
4888	/tmp/XMRIG	55
Sample File 14 real_live		
4898	/tmp/real_live	30
Sample File 15 help		
4987	/tmp/help	55
Sample File 16 view_list		
4590	/tmp/view_list	11
Sample File 17 unlucky		
4592	/tmp/unlucky	60
Sample File 18 keepass		
4000	/tmp/keepass	1
Sample File 19 harden.bin		
4792	/tmp/harden.bin	3
Sample File 20 npee		
4789	/tmp/npee	10
Sample File 21 abc.tar		
4772	/tmp/abc.tar	5

Sample File 22 unknown	
4789	/tmp/unknown	3

Sample File 2 process IDs, file path, and timely execution do not give enough information. The process IDs only indicates the association of the process with the path. The process IDs 4597 and 4598 create the new fork and execute the /net/route command each time. The excessive use of /net/route process as noticed in the sample 2 execution identifies that the Remote discovery technique is used (Stepanic, 2017).

As shown in Figure 5.4, the process opened the path /proc/4582/net/route multiple times during the execution of the malicious sample file 2. The process ID 4582 opened multiple processes that run/net/route while utilisation reached approximately 90%. Therefore, the data shows that the Table 5.4 sample test with file two can identify the malicious activity with the combined activity of CPU usage, process utilisation, and execution count. It leads to the detection of the zero-day malware with the anomaly behaviour detected by the process. The only difference compared to sample 1 is that the process IDs do not directly indicate the abnormal behaviour but the fork processes that initiate the /net/route. The /net/route process can be used in the normal situation; however, the execution with that large number and the high CPU utilisation marked this file as malicious.

Sample file three, same as sample file two, does not give enough information when executed. The execution indicates that the malicious software is trying to get the remote discovery of the system and the masquerading and non-standard port for command and control tactics of an attack. The remote discovery enables the lateral movement such that malicious actors can hop from the compromised machine to the other machine in the network. The malicious software sends the traffic to the public IP address on a non-standard port. From the details collected from process IDs, the type of attack detected in the remote discovery. The process IDs only indicates the association of the process with the path. The process IDs 4596 and 4597 creates the new fork and execute the /net/route

command each time. As shown in figure 5.4, process ID 4582 opened multiple processes that run /net/route while utilisation reached approximately 100%.

Therefore, the data shows that the Table 5.4 sample test with file two can identify the malicious activity with the combined activity of CPU usage, process utilisation, and execution count. It leads to the detection of the zero-day malware with the anomaly behaviour detected by the process. The only difference compared to sample 1 is that the process IDs do not directly indicate the abnormal behaviour but the fork processes that initiate the /net/route. It is the same behaviour as seen in the sample file two execution. The /net/route process can be used in the normal situation; however, the execution with that large number and the high CPU utilisation marked this file as malicious.

The sample file four execution initiates the process ID 4583, which results in creating multiple new forks. The process ID 4592, 4593 and 4595 enables the /net/route, enabling remote discovery. The process ID 4598 along with /net/route opens the /dev/watchdogs, which supports the remote discovery of the system information. To disguise the operation, the process IDs 4596 shows that the masquerading is performed with the /proc/410/maps, which hides the malicious software to be executed in the legitimate program, as shown in figure 4.30. The route, watchdog, and map can be executed in normal circumstances, but the combination of the processes, CPU utilisation, and event count treated the file as malicious activity.

Overall results from the study indicate that the system's process utilisation can detect malicious activities. There are no specific processes to detect zero-day malware. However, the processes which lead to access different file execution as sample1 file processes use ABRT and the libre report with excessive usage which impacts CPU utilisation, sample file 2 and 3 uses /net/route to executes the remote discovery and sample file 4 runs the /net/route, watchdog and maps. The impact of the processes triggers the malicious activity. The usage of the processes initiated by the files identifies the file as malware.

Unexpected results were detected in two sample files, numbers 9 and 20. It was observed that sample file 9 was a confirmed malicious file, as per multiple threat intelligent repositories, but it did not fall under the abnormal file behaviour. The same result was noticed when the confirmed non-malicious sample file 20 was executed. The utilisation and the process count went high for this non-malicious file. The reason for the failed results was due to the detection criteria of the abnormal behaviour. For the lab setup, the abnormal behaviour was based on the combination of CPU usage and file execution count, but these two sample files did not follow the expected pattern.

The above discussion derived from the test results in chapter 4 as designed in the setup explained in chapter 3 shows that the system can detect zero-day malware by observing the anomaly behaviour. The detection can lead to contains zero-day malware. It was also observed that a robust mechanism is required to monitor the processes to avoid false positives.

Question 2 (Q2)- How can YARA rules and antivirus software be integrated after the zero-day malware is detected?

Answer:

The detection of the malicious file leads to classifying the file with the information used to write the relevant YARA rules. The YARA rules can be integrated with the end-point protection tool using a programmable application interface to update the signatures. The programmable interface can enable the scan to remove the zero-day malicious file. The observation assures that the programmable interface automates the containment of the malicious file.

Discussion:

This research question looks for what method is used to remediate the malicious file after such a file is detected. The malicious file detection is performed by the method presented in the above section in answering question 1. The integration involves the programable

interface between the anti-virus tool and the program which writes the YARA rule. The solution is to write the YARA tool and the procedure to update the anti-virus signature locally on the client machine, as shown in figure 3.3. The first step after identifying the file is to take the information of the file, which can be used to identify it. The method used for the file classification is the hex dump that runs using a programmable interface which calls the Python script. The magic number of the file and other random hex values of the files are used. The information is then called in writing the YARA rules. The steps explained in the coming section for the top 4 malware are performed automatically without any manual changes.

Table 5.5 shows the summarized sample YARA rules configured for the first 4 sample files. The variable information $a, $b and $c is extracted from the hex dump tool, referenced in Figures 4.10, 4.17, 4.24 and 4.31, respectively.

Table 5. 5 - *Sample YARA Rule*

Sample 1	Sample 2	Sample 3	Sample 4

{
Meta:
desc = "Sample 1 virus file"
strings:
$a = {7f 45 4c 46 02 01 01 00}
$b= {07 78 49 4d 20 55 7a 49}
$c= {69 31 f6 fd 24 0d 2a 2e}
Condition:
$a and $b and $c
}
	{
	Meta:
	desc = "Sample 2 virus file"
	strings:
	$a = {7f 45 4c 46 02 01 01 00}
	$b={5f 5f 47 49 5f 5f 5f 65}
	$c={74 73 69 64 00 5f 5f 47}
	Condition:
	$a and $b and $c
	}
		{
		Meta:
		desc = "Sample 3 virus file"
		strings:
		$a = {7f 45 4c 46 02 01 01 00}
		$b= {6e 65 74 5f 60 64 64 72}
		$c= {6e 61 6d 65 00 63 6c 6f}
		Condition:
		$a and $b and $c
		}
			{
			Meta:
			desc = "Sample 4 virus file"
			strings:
			$a = {7f 45 4c 46 02 01 01 00}
			$b= {6c 30 30 64 00 66 63 6e}
			$c= {70 65 5f 74 6f 6c 6f 77}
			Condition:
			$a and $b and $c
			}

The traditional malware detection tool detects the malicious file based on the availability of the signatures (Christodorescu et al., 2007), which are prepared by the threat intelligence by doing the sandboxing of the file (Joseph & Mukesh, 2019). It updates the signature database of the antivirus tool. However, the zero-day malware will not have the signature updated when detected the first time. The method explained in chapter 3 also relies on the signature, which is updated using the YARA rule; however, the file detection is detected by the abnormal behaviour of the system doing the malicious file execution. The research shows that rather than sandbox detects the malicious file the client installed on the user machine integrated by the API using the Python in the lab setup can write the signatures and update them on the end-point antivirus tool.

As highlighted the challenge in the literature review section, malware analyst uses the YARA rule for threat hunting purposes. It allows them to create the YARA rule conditions manually and then match them against the malicious files. The process is slow as all the work is manual, whereas the technique explained in chapter 3 enables automation. This existing technique is part of the static malware analysis where the sample malware is examined without the sample file execution and then hash values updates on the threat intelligent system. The information is then used by the end-point protection tool to act on the file accordingly. Chapter 4 results confirm the use of the hex dump to identify the file

type and use specific parameters in Table 5.3 for the file identification. A similar challenge is faced in the case of existing dynamic analysis. The discovery of the file is taken place by executing the file in a sandbox environment. The malicious activity marking it as malware, the file identification is again used to write the YARA rules, which manually helps in either writing the antivirus signature or finding the malicious file manually on the system. Compared to the static and dynamic analysis and the file detection and action method, the approach explained in chapter 3 is fast to react due to the automated API integration with the end-point antivirus tool. The steps explained in chapter 3 after the file identification include the malicious file classification, writing the YARA rules using the file classification details, integrating with the end-point antivirus tool, and using the API call to execute the scan to detect and quarantine the malicious file. Python has been used as the application programming interface to execute the list of steps mentioned above. Other programming tools, node js and ruby language, have been used for the API for the end-point antivirus tool, but these have slower responses than Python. They have more computational requirements and security issues. (Salterwaterc, 2021). The Python library for ClamAV integration gives more flexibility in terms of usage and the available module as per the required testing in chapter 4 (Grainger, 2021). The main advantage discovered during the lab setup is the availability of the libraries for Python. The usage has been fast in responsiveness and less memory intensive. The significant difference with the YARA rule usage with static and dynamic file analysis is detecting the zero-day malicious code. The time utilized by the static and dynamic analysis technique and the manual effort for the file search does not contain the zero-day malicious file. In contrast, the proposed setup results in chapter 4 use the abnormal file behaviour by executing the file and marking the file as malicious. Based on unusual behaviour, in this case, as in chapter 4, high CPU utilization and the count detected files can be automatically searched via API and quarantined..

5.3. Automate Threat Intelligence

The flow for detecting the zero-day malicious file also specifies the action to update the virustotal threat intelligence if the file is malicious. The test results have used the existing known malware, known to the virustotal threat intelligence as mentioned in section 5.2; therefore, the step is not included in the chapter 4 test results. However, the thesis gives the functionality to automatically update the virustotal threat intelligence using the programable interface and Python libraries. As per the other techniques mentioned in chapter 2, the malware analysis techniques have been used to update the signature after doing static or dynamic malware analysis, which updates the threat intelligence site, including virustotal. The study shows the framework can be integrated with multiple components to automate the process from detecting the malicious file execution based on the processor usage. The process usage can cause anomaly activities on the system, which can vary. In our lab set up in chapter 4, the abnormal behaviour comprised the high resource usage and the execution of files not required in a typical scenario. The first phase of detecting the file and the second phase action on the detected file can be automated to quarantine zero-day malware.

5.4. Conclusion

Chapter 5 has discussed the result collected from the testing as completed in chapter 4. It also answered the research questions posed by this study. The result depicts the method and technique to detect zero-day malicious software. The method to integrate the YARA rules and the end-point protection was also discussed. The sample malware used to provide the test results in chapter 4 uses the setup as discussed in chapter 3. The chapter concludes the discussion by detecting the abnormal activity on the Linux system for zero-day malware and displays the technique using an application

programable interface to update the YARA rule and end-point anti-virus signature to remove the zero-day malware.

6. Conclusion

This chapter concludes the discussion on the technique to evaluate a process to detect zero-day malware files. The following section gives a brief overview of the previous chapters. The following section highlight approach's contribution, followed by discussing some of the limitations of the research that could have caused some change in the result. The last section of the chapter discusses the activities for the following up of this work.

6.1. Summary of Research

Chapter 1 introduced the topic of zero-day malware detection from the abnormal behaviour of the process when the malicious file is executed. It briefly overviews the current challenges in the existing system to protect end-point from malicious software. It also provides an initial discussion about the zero-day malware and the known malicious file, which has gone through the proper analysis. It also highlighted the structure of the thesis work and set the base for the motivation behind this work. Chapter 2 details the literature review and the current challenges and security concerns while handling the current mode of controlling zero-day malware on the Linux systems. It also deep dive into the architecture of the Linux operating system and its underlying concepts. It covered multiple process calls, including forks, memory architecture and briefly covered the forensic artefacts. Chapter 2 also discussed the malware and its analysis techniques. The current signature-based approach for the end-point protection was detected, and the cyber threat intelligence role was overviewed. Chapter 2 ends with the current technology for zero-day malware detection and the challenges highlighted in the current model. It set the foundation to raise the research questions. The research questions were driven by the challenges and concerns as indicated in chapter 2.

The study of similar research work and industry-based framework was assessed to define the methodology and the design. The research design in chapter 3 describes the

method and technique. It reviews the procedure, malware source and the test setup environment. Chapter 3 provides the details for setting up the environment. The approach defined the test flow and defined the lab setup environment to answers the research question.

In chapter 4, the test results were presented based on the approach defined in chapter 3. Those test results demonstrated the expected outcome in a controlled environment. The result analysis and the finding were discussed in chapter 5. It continued to link the literature review and explained the answers to the research questions.

6.2. Contribution

The thesis discusses the method to resolve the problem due to the unavailability of signatures for the zero-day malware. Instead of sandboxing the malware for further analysis, the technique monitors the operating system malicious file's abnormal activity and identifies the malware on the system. The deviation of the monitoring data during the normal situation and at the time of malicious file execution and the processes utilisation triggers the malicious file and its location on the system. The second phase of the thesis covers the procedure to remove the malicious file after detection using multiple programable interfaces. The malicious file signature is written after detecting the abnormal behaviour method using YARA rules which automatically remove the malware from the system by integrating with the anti-virus tool.

6.3. Limitation

This section discusses some of the items which have impacted the outcome of the research. Some of the factors might have changed the result and the techniques which could have been improved.

6.4.1 System Requirements

The client endpoint is based on the specific distribution of the Linux operating system. Different distribution has different system files and libraries, which can impact the detection mechanism of the processes that generate abnormal behaviour on the system. Python program is written based on the operating system's libraries and the dependency limits the scope of the testing. The programmable interfaces will vary depending on the CPU architecture as different CPUs have different opcodes for the instruction set. Hence the bytes to be detected in the YARA rule might be different. The programmable interfaces created for the monitoring of the processes on the systems would have been different. The coding language has to use different libraries for the integration to collect the information as different versions of Operating System may use different system libraries. If multiple distributions were used and variations of programmable interfaces written in multiple programming languages, then detecting the zero-day malware based on the abnormal process execution would have been a more successful result.

6.4.2 End-Point Protection

The open-source version of end-point protection is used, which supported the programable interface to integrate signature updates on the anti-virus installed on the client machine. The YARA rule used on the end-point has a simplistic approach and might have impacted complex file types. The configuration of the programable interface to push the signatures generated by the YARA rule to automate the process to remove the malicious file would vary on the supported version of the end-point. It has a dependency on the available supporting library of the programming language used to write the API. The open-source version supported removing the available signature set from the client machine, which enabled the test setup. The lab environment would not have provided the test result with the existing sample malware data set if the client end-

point would not have allowed updating the signature. However, the other option of honey-pot setup to collect the malicious malware would have required a strict environment.

6.4.3 Sample Malware for Data Set

A different set of malwares might have impacted the outcome of the test results. The selected malware resulted in the system's high CPU utilisation and the execution of multiple processes. The result was tested only on the limited numbers of sample files. More samples would have provided a different variation of how they operate and might have impacted the test result outcome. The malware selected in the test setup is known malware and is used in the test setup by disabling the signatures on the existing anti-virus tool. The malware used was not zero-day malware as the signatures were already available. Due to this reason, a programable interface to update the virustotal could not be tested. The sample malware is also limited to the specific operating system architecture, and different variations would have generated different results..

6.4.4. Detection Criteria

The detection criteria to monitor the processor utilization does not include the machine learning. The lab setup monitors the CPU usage and the number of times the file is executed to detect abnormal behaviour. Even though two results failed from the expected detection behaviour, the result might have been different with more tests. Also, considering the memory utilization as the detection criteria and the CPU would have impacted the lab results. However, the detection process would have been more effective if the detection criteria included the combined monitoring of CPU, memory, and machine learning component.

6.4. Future Work

 Different areas can be driven forward from this research. One of the areas is IoT systems security. IoT systems architecture needs to be explored to detect abnormal malware activities. The research will also be required to understand the malware behaviour for a different variation of IoT architecture. The abnormal behaviour on the system processes can impact network performance for the devices connected to the same network. Hence, the thesis can also enhance the work on analysing abnormal network behaviour if multiple systems are involved in an abnormal activity.

The other area which can be focused on is the study of false-positive detection using the approach for detecting and removing the malicious file based on the abnormal behaviour. The action taken is based on the active device and not on the sandbox. This work will strengthen the method and technique used in this research.

Multiple programmable interfaces are configured to perform the different tasks at different stages. A work to consolidate the diverse task in one client agent can consolidate and increase the efficiency of the method.

The chapter concludes that the detection of zero-day malware is essential with the emerge of new techniques to exploit the vulnerabilities of the Linux based operating system. The sample malware was tested on the active client machine, which was not sandboxed, and observed abnormal behaviour. The deviation of the client machine from the expected behaviour detects the malicious file. The detection is also based on the processor utilization and the undesired process file execution. The detection of the file and updating the end-point protection to remove the malicious file also enables the automated process.

Total twenty-two sample malware was used to test the method and technique to detect the zero-day malware activity, and most of the time, the system was able to detect the malicious file. The method and technique defined in chapter 3 can determine the zero-

day malware by robust monitoring detection processes to identify abnormal behaviour, which can be automated to remove the malicious file.

References

Akram, B., & Ogi, D. (2020). The Making of Indicator of Compromise using Malware Reverse Engineering Techniques. *7th International Conference on ICT for Smart Society: A IoT for Smart Society, ICISS 2020 - Proceeding*, 2–7. https://doi.org/10.1109/ICISS50791.2020.9307581

Alvarez, V. M. (2021). *VirusTotal/ yara-python*. Github. Retrieved from https://github.com/VirusTotal/yara-python

Stepanic, D. (2017). *Remote System Discovery*. Mitre Attack. Retrieved from https://attack.mitre.org/techniques/T1018/

Baker, K. (2019). *What is Cyber Threat Intelligence? [Beginner's Guide]*. Crowdstrike. Retreived from https://www.crowdstrike.com/epp-101/threat-intelligence/

Boukhtouta, A., Mokhov, S. A., Lakhdari, N. E., Debbabi, M., & Paquet, J. (2016). Network malware classification comparison using DPI and flow packet headers. *Journal of Computer Virology and Hacking Techniques*, *12*(2), 69–100. https://doi.org/10.1007/s11416-015-0247-x

Carrigan, T. (2020). *A beginner's guide to firewalld in Linux*. Redhat. Retrieved from https://www.redhat.com/sysadmin/beginners-guide-firewalld

CentOS-Org. (2021). *CentOS*. Retrieved from https://www.centos.org/about/

Chen, X., Sha, E. H. M., Jiang, W., Zhuge, Q., Chen, J., Qin, J., & Zeng, Y. (2016). The design of an efficient swap mechanism for hybrid DRAM-NVM systems. *Proceedings of the 13th International Conference on Embedded Software, EMSOFT 2016*. https://doi.org/10.1145/2968478.2968497

Christodorescu, M., Jha, S., Maughan, D., Song, D., & Wang, C. (2007). Malware Detection [book]. In *Journal of Chemical Information and Modeling* (Vol. 53, Issue 9).

Ciancioso, R., Budhwa, D., & Hayajneh, T. (2018). A framework for zero day exploit detection and containment. *Proceedings - 2017 IEEE 15th International Conference on Dependable, Autonomic and Secure Computing, 2017 IEEE 15th International Conference on Pervasive Intelligence and Computing, 2017 IEEE 3rd International Conference on Big Data Intelligence and Compu, 2018-Janua*, 663–668. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.116

Clam-AntiVirus. (2020). *ClamAV*. Retrieved from https://www.clamav.net/documents/introduction

Clamav-signature-update. (2020). *Creating signatures for ClamAV*. ClamAV. Retrieved from https://www.clamav.net/documents/creating-signatures-for-clamav#creating-signatures-for-clamav

Comar, P. M., Liu, L., Saha, S., Tan, P. N., & Nucci, A. (2013). Combining supervised and unsupervised learning for zero-day malware detection. *Proceedings - IEEE INFOCOM*, 2022–2030. https://doi.org/10.1109/INFCOM.2013.6567003

Cozzi, E., Graziano, M., Fratantonio, Y., & Balzarotti, D. (2018). Understanding Linux Malware. *Proceedings - IEEE Symposium on Security and Privacy*, *2018-May*, 161–175. https://doi.org/10.1109/SP.2018.00054

Crowdstrike. (2021). Zero-day Attack Explained. Crowdstrike. Retrieved from https://www.crowdstrike.com

Damri, G., & Vidyarthi, D. (2016). *Automatic Dynamic Malware Analysis Techniques For Linux Environment.* *67*, 825–830.

Dmitry, M., & Elena, P. (2020). Linux Privilege Increase Threat Analysis. *Proceedings - 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology, USBEREIT 2020*, 579–581. https://doi.org/10.1109/USBEREIT48449.2020.9117739

Engel, D. (2017). *Configure development environment for pymssql Pythondevelopment.* https://docs.microsoft.com/en-us/sql/connect/python/pymssql/step-1-configure-development-environment-for-pymssql-python-development?view=sql-server-ver15

Ewais, M. A., Omran, M. A., Raafat, A., & Alkabani, Y. (2016). A virtual memory architecture to enhance STT-RAM performance as main memory. *Canadian Conference on Electrical and Computer Engineering*, *2016-Octob*. https://doi.org/10.1109/CCECE.2016.7726657

Fox, R. (2014). Linux with Operating System Concepts. In *Linux with Operating System Concepts.* https://doi.org/10.1201/b17269

Gandotra, E., Bansal, D., & Sofat, S. (2017). Zero-day malware detection. *Proceedings - 2016 6th International Symposium on Embedded Computing and System Design, ISED 2016*, 171–175. https://doi.org/10.1109/ISED.2016.7977076

Gao, Y., Hasegawa, H., Yamaguchi, Y., & Shimada, H. (2021). Malware Detection Using Gradient Boosting Decision Trees with Customized Log Loss Function. *International Conference on Information Networking*, *2021-Janua*, 273–278. https://doi.org/10.1109/ICOIN50884.2021.9333999

Grainger, T. (2021.). Clamd 1.0.2. pypi. Retrieved from https://pypi.org

Jicha, A., Patton, M., & Chen, H. (2016). SCADA honeypots: An in-depth analysis of Conpot. *IEEE International Conference on Intelligence and Security Informatics: Cybersecurity and Big Data, ISI 2016*, 196–198. https://doi.org/10.1109/ISI.2016.7745468

Joseph, L., & Mukesh, R. (2019). To Detect Malware attacks for an Autonomic Self-Heal Approach of Virtual Machines in Cloud Computing. *5th International Conference on Science Technology Engineering and Mathematics, ICONSTEM 2019*, 220–231. https://doi.org/10.1109/ICONSTEM.2019.8918909

Kara, I. (2019). A basic malware analysis method. *Computer Fraud and Security*, *2019*(6), 11–19. https://doi.org/10.1016/S1361-3723(19)30064-8

Kerrisk, M. (2010). *The Linux Programming Interface.*

Kessler, G. (2021). GCK File Signature Table. Garykessler. Retrieved from

https://www.garykessler.net

Kim, J., Min, C., Kim, J., Kang, D. H., Kim, I., & Eom, Y. I. (2014). Page allocation scheme for anti-fragmentation on smart devices. *2014 IEEE 3rd Global Conference on Consumer Electronics, GCCE 2014*, 512–513. https://doi.org/10.1109/GCCE.2014.7031168

Klaus, T., & Elzweig, B. (2020). The impact of data breaches on corporations and the status of potential regulation and litigation. *Law and Financial Markets Review*, *14*(4), 255–260. https://doi.org/10.1080/17521440.2020.1833432

Ko, S., Jun, S., Ryu, Y., Kwon, O., & Koh, K. (2008). A new Linux swap system for flash memory storage devices. *Proceedings - The International Conference on Computational Sciences and Its Applications, ICCSA 2008*, 151–156. https://doi.org/10.1109/ICCSA.2008.54

Li, M., & Liu, J. (2017). *How can Advanced Sandboxing Techniques Thwart Elusive Malware?* Trend Micro Security News. https://www.trendmicro.com/vinfo/us/security/news/security-technology/how-can-advanced-sandboxing-techniques-thwart-elusive-malware

Liu, L., Li, Y., Ding, C., Yang, H., & Wu, C. (2016). Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? *IEEE Transactions on Computers*, *65*(6), 1921–1935. https://doi.org/10.1109/TC.2015.2462813

Love, R. (2010). *Linux Kernel Development*.

Lungana-Niculescu, A. M., Colesa, A., & Oprisa, C. (2018). False positive mitigation in behavioral malware detection using deep learning. *Proceedings - 2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing, ICCP 2018*, 197–203. https://doi.org/10.1109/ICCP.2018.8516611

Martin, I., Hernandez, J. A., De Los Santos, S., & Guzman, A. (2018). Analysis and evaluation of antivirus engines in detecting android malware: A data analytics approach. *Proceedings - 2018 European Intelligence and Security Informatics Conference, EISIC 2018*, 7–14. https://doi.org/10.1109/EISIC.2018.00010

Mcafee. (2021). What Is Endpoint Security?. Mcafee. Retrieved from https://www.mcafee.com/

Mehdi, S.B., Tanwani, A.K., Farooq, M. (2009). *IMAD: In-execution malware analysis and detection*.

Mohanta, A., Saldanha, A., Mohanta, A., & Saldanha, A. (2020). Antivirus Engines. In *Malware Analysis and Detection Engineering*. https://doi.org/10.1007/978-1-4842-6193-4_22

MySQL-Dev. (2021). *MySQL Community Downloads*. Retreived from https://dev.mysql.com/downloads/repo/yum/

Naik, N., Jenkins, P., Cooke, R., Gillett, J., & Jin, Y. (2020). Evaluating Automatically Generated YARA Rules and Enhancing Their Effectiveness. *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*, 1146–1153. https://doi.org/10.1109/SSCI47803.2020.9308179

Naik, N., Jenkins, P., Savage, N., Yang, L., Naik, K., Song, J., Boongoen, T., & Iam-On, N. (2020). Fuzzy Hashing Aided Enhanced YARA Rules for Malware Triaging.

2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020, 1138–1145. https://doi.org/10.1109/SSCI47803.2020.9308189

Nothaas, S., Beineke, K., & Schoettner, M. (2019). Optimized memory management for a Java-based distributed in-memory system. *Proceedings - 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, 668–677. https://doi.org/10.1109/CCGRID.2019.00086

Or-Meir, O., Nissim, N., Elovici, Y., & Rokach, L. (2019). Dynamic malware analysis in the modern era—A state of the art survey. *ACM Computing Surveys*, *52*(5). https://doi.org/10.1145/3329786

Pypi-org. (2020). *psutil.* https://pypi.org/project/psutil/

Radhakrishnan, K., Menon, R. R., & Nath, H. V. (2019). A survey of zero-day malware attacks and its detection methodology. *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, *2019-Octob*, 533–539. https://doi.org/10.1109/TENCON.2019.8929620

Salterwaterc, K. (2021.). *NodeJS Clamscan Virus Scanning Utility*. Npmjs. Retrieve from https://www.npmjs.com

Shah, K., & Singh, D. K. (2016). A survey on data mining approaches for dynamic analysis of malwares. *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015*, 495–499. https://doi.org/10.1109/ICGCIoT.2015.7380515

Shahzad, F., Bhatti, S., Shahzad, M., & Farooq, M. (2011). In-execution malware detection using task structures of Linux processes. *IEEE International Conference on Communications*, 0–5. https://doi.org/10.1109/icc.2011.5963012

Shahzad, F., Shahzad, M., & Farooq, M. (2013). In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. *Information Sciences*, *231*, 45–63. https://doi.org/10.1016/j.ins.2011.09.016

Shipp, R. (2020). *Online Scanners and Sandboxes.* https://github.com/rshipp/awesome-malware-analysis#online-scanners-and-sandboxes

Shotts Jr, W. E. (2012). *The linux command line: {A} complete introduction*.

Stazi, G., Menichelli, F., Mastrandrea, A., & Olivieri, M. (2017). Introducing approximate memory support in Linux Kernel. *PRIME 2017 - 13th Conference on PhD Research in Microelectronics and Electronics, Proceedings*, 97–100. https://doi.org/10.1109/PRIME.2017.7974116

Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems. In *Education* (Vol. 2). http://www.amazon.com/dp/0136006639

The-Talos-Group-AT-Cisco. (2018). *ClamAV Signature Creator*. Retrieved from https://github.com/Cisco-Talos/CASC

Vasilescu, M., Gheorghe, L., & Tapus, N. (2014). Practical malware analysis based on sandboxing. *Proceedings - RoEduNet IEEE International Conference*, 1–6. https://doi.org/10.1109/RoEduNet-RENAM.2014.6955304

Virus-Share. (2020). *Virus Share.* https://virusshare.com/

Virus-total. (2021). *VirusTotal Reports*. Retrieved from
https://support.virustotal.com/hc/en-us/articles/115002719069-Reports

Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D. A., & Aigner, W.
(2015). A Survey of Visualization Systems for Malware Analysis. *Eurographics
Conference on Visualization (EuroVis)*, 105–125.
http://mc.fhstp.ac.at/sites/default/files/publications/wagner_2015_eurovis_star_mal
warevis_postprint_reduced.pdf

Wu, K., Ge, Y., Chen, W., & Zhang, T. (2012). The research and implementation of the
Linux process real-time monitoring technology. *Proceedings - 4th International
Conference on Computational and Information Sciences, ICCIS 2012*, 1046–1049.
https://doi.org/10.1109/ICCIS.2012.342

Yararules. (2021). *Yara Rules Project*. Retrieved from https://github.com/Yara-Rules/

Yaswinski, M. R., Chowdhury, M. M., & Jochen, M. (2019). Linux security: A survey. *IEEE
International Conference on Electro Information Technology*, *2019-May*, 357–362.
https://doi.org/10.1109/EIT.2019.8834112

Yildiz Çavdar, Z., AVCI, İ., KOCA, M., & SERTBAŞ, A. (2019). A Survey of Hybrid Main
Memory Architectures. *Sakarya University Journal of Science*, 1–1.
https://doi.org/10.16984/saufenbilder.334645

Zeltser, L. (2021). *Free Automated Malware Analysis Sandboxes and Services*.

Kizza, J., & Migga Kizza, F. (2011). Intrusion Detection and Prevention Systems.
Securing the Information Infrastructure, 239–258. https://doi.org/10.4018/978-1-
59904-379-1.ch012