Evaluating Open Source Malware Sandboxes with Linux malware

OLABOYEJO OLOWOYEYE

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

2018

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 acknowledgment is made in the acknowledgments.

----------  ----------

OLABOYEJO OLOWOYEYE

(05-June-2018)

Abstract

Analysis of Linux binaries for indicators of compromise is an area of research gaining in interest due to the ubiquity of Internet connected embedded devices. These devices have also been the subject of high profile cybersecurity incidents as a result of the damage caused by their compromise. Malware analysis sandboxes are used to examine malware samples in an isolated environment. They provide a safe environment for the analysis of malware. Most of the discussion on malware analysis and associated tools have been devoted to the Windows operating system. This is because the Windows operating system is the dominant operating system in the desktop operating system space. This research examines the Linux operating system and evaluates the malware analysis sandboxes that are available for the examination of malware developed for the platform. These analysis sandboxes were tested against Linux malware binaries and the relative effectiveness of the sandboxes were observed.

Malware samples were sourced from online repositories and a honeypot setup. The malware samples obtained from the repositories were restricted to those first submitted to the portals within the last four years. The honeypot was deployed to attract malware samples in the wild that are possibly unknown to existing portals. Four malware samples were extracted from the honeypot which were added to the two hundred and ninety-three (293) selected from VirusTotal and VirusShare. The five sandboxes tested were REMnux, Limon, Cuckoo, Detux and HaboMalhunter. The malware samples were examined and analysed on these platforms. The static and dynamic analysis features of these tools were observed as well as their support for automation and reporting. The consistency of the results where applicable were also noted.

It was observed that despite the consistency of analysis noticed; collectively, the five sandboxes failed to detect indications of compromise in twenty-seven (27) of two hundred and ninety-seven (297) malware samples. HaboMalhunter was found to be the most effective during dynamic analysis in the detection of indications of compromise; however, its workflow required each analysis run to be done manually because it did not have in-built virtual machine orchestration like Limon, Detux and Cuckoo. During static analysis results, the results were observed to be similar with the exception of Limon which employed Yara rules to detect the packers used to mask the malware samples. Limon was also alone in its use of Context Triggered Piecewise Hashing (CTPH) to determine the similarity between malware samples by its maintenance of a master list of analysed samples. Cuckoo and HaboMalHunter generated output reports in HTML and JSON while Detux supported only JSON output. REMnux and Limon generated only plaintext output reports. The addition of virtual machine control to HaboMalhunter to restore virtual machine state before and after each analysis run was suggested as a recommended improvement to facilitate the automation of the analysis process. The need to develop more packing signatures for Yara rules was also mentioned for the automatic detection of packers.

Contents

List of Figures

List of Tables

List of Abbreviations

AEMS	Analysis Evasion Malware Sandbox
AIDE	Advanced Intrusion Detection Environment
ASCII	American Standard Code for Information Interchange
BIOS	Basic Input Output Systems
CFQ	Completely Fair Queuing
CFS	Completely Fair Scheduler
CTPH	Context Triggered Piecewise Hashing
DMA	Direct Memory Access
DNS	Domain Name System
ELF	Executable and Linkable Format
FIFO	First in First out
GOT	Global Offsets Table
HCI	Human Computer Interaction
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IEEE	Institute of Electronic and Electrical Engineers
IoT	Internet of Things
IRC	Internet Relay Chat
ISA	Industry Standard Architecture
JSON	JavaScript Object Notation
LTS	Long-Term Support
MBR	Master Boot Record
MMU	Memory Management Unit

PDF	Portable Data Format
PFN	Page Frame Number
PHT	Program Header Table
PID	Process Identifier
PLT	Procedure Linkage Table
POSIX	Portable Operating System Interface
SHT	Section Header Table
SSH	Secure Shell
SVM	Support Vector Machine
TID	Task Identifier
TLB	Translation Look-aside Buffer
TLSH	Trend Locality Sensitive Hashing
URL	Uniform Resource Locator
VFS	Virtual File System
VLAN	Virtual LAN
VPS	Virtual Private Server

1. Introduction

1.1 Background, motivation and objective

Malicious software (or malware) are programs written with the intention of causing harm to the target of the program execution (Moser, Kruegel, & Kirda, 2007). The ubiquity of the Internet has helped facilitate the spread of malware infections. Business enhancement tools such as instant messaging, electronic mail and shared files which have become an essential part of collaboration are being used as enablers for the spread of malware in the enterprise. Critical infrastructure services in health, transportation, energy and communication are supported by Information technology systems. The potential impact of harmful tools on these services have not gone unnoticed to criminal organisations. Criminal activities using malware have ranged from the deployment of keyloggers and spyware to steal data from individual users to corporate theft, blackmail and sabotage. A couple of high profile attacks against critical infrastructure are the Wannacry ransomware attack that particularly affected the operations of the National Health Service in the UK and the *Trojan.Disakill* attack on the Ukraine energy infrastructure (Ehrenfeld, 2017; Symantec, 2017).

The most popular platforms, due to the large size of infection footprint generally have the most malware attacks and hence the most amount of analysis activities and tools. The foregoing is the reason that the erroneous view was held that there were no viruses on Linux. The computer desktop market is dominated by the Microsoft Windows operating systems. The uptake in the use of Linux for servers and the increasing popularity of Internet connected embedded systems and the Internet of Things (IoT) have suddenly made the Linux operating system a lucrative target. More than half of the malware samples attracted by the Symantec IoT honeypot were written for the Linux operating system as vulnerabilities on the Linux operating system are being actively sought for exploits (Symantec, 2018a). The *Trojan.Disakill* attack on the Ukraine energy infrastructure in 2016 was a disk wiping attack on Linux servers supporting the energy grid. One of the highest profile exploits was directed at the web hosting company OVH, it involved the compromise and enlistment of IoT devices into a botnet using the Mirai malware to create one of the biggest Distributed Denial of Service attacks (Symantec, 2017).

Malware analysis allows researchers to dissect malware to determine their objectives and operations. This activity can be useful in creating static, behavioural and heuristic signatures that can be added to security appliances. It can also be used to stop the effect of an ongoing attack as well as for conducting a post-mortem analysis of a breach. Malware analysis sandboxes allow execution and examination of the malware samples in a safe and isolated environment. Malware samples are also released at a very high rate that challenges the ability of the analyst and the available tools to cope. It has been found that most new malware samples are variants of existing ones. In order to cope with the rate of malware deployment, the ability to detect variations of existing samples is necessary as is the ability to automate analysis in a safe environment.

The foregoing motivations are due to the prevalence of malware and the increasing interest in Linux malware samples by malware authors as well as the need for automated analysis in a safe environment. The analysis of Windows operating system based malware has been given the most coverage in available literature (Botacin, de Geus, & Grégio, 2017). This research seeks to explore and evaluate the existing tools and platforms for malware analysis on Linux systems. This involves the sourcing of malware samples as well as the tooling environment. This appraisal is to guide the security

community on the relative utility of the available tools for malware analysis in the Linux environment and recommendations to the open source community on improvements to these tools.

1.2 Organisation

The thesis is structured into six parts. The first part is the introduction which discussed the background and motivation for the thesis. The underlying concepts involved with internals of the Linux operating system and malware analysis were addressed in the second chapter as well as the related research activities. The five sandboxes, REMnux, Limon, Cuckoo, Limon and HaboMalHunter were selected for testing after the review. The third chapter explored the methods used for sourcing the malware samples and the research design. Drawing from existing research the malware samples were obtained from malware repositories and a honeypot setup. The honeypot setup was described and testing methodology for the five sandboxes decided upon in chapter 2, laid out. The research sub-questions and hypotheses concluded the chapter. The result of the honeypot entrapment scheme and the malware analysis results on the sandboxes were presented in chapter 4. The research sub-questions related to automation and reporting features were also answered in chapter 4. Chapter 5 was a discussion of the results within the context of the malware families and the CPU families of the malware samples. The remaining research sub-questions and the research hypotheses were answered in chapter 5. Chapter 6 concluded the thesis with an overview, the contribution and the limitations of the research. Suggestions for future work were also made.

2 Literature Review

2.1 Introduction

This chapter is a review of the existing body of work upon which this research builds. It is divided into five sections. The first section discusses malware classification and analysis with respect to the Linux operating system. A study of malware analysis on the Linux operating system requires a thorough understanding of the internal workings of the operating system. The second section accomplishes this with a discussion of internals of the Linux operating systems, with brief descriptions of some key system calls and forensic artefacts, while the third explores the subject of malware analysis techniques. A review of similar research activities was undertaken in the fourth section and the chapter concludes with the identification of the gaps this research seeks to fill in light of the review.

2.2 Malware and the Linux Operating System

Malware is software designed with harmful intent, affecting optimal and secure operation of a computing environment (Bryant, 2016). Malware can be categorised based on mobility, that is, if it can spread without human interaction. Viruses, worms and mobile code are examples of mobile malware while Trojans and rootkits are non-mobile malware (Boyle & Panko, 2014). Viruses are self-propagating malicious code that are set in motion by execution of legitimate benign programs which serve as hosts to the viruses (Szor, 2005). Viruses enter a system through several means. Computers ship with Basic Input Output Systems (BIOS) on the motherboards. These are generic instructions with as little assumptions as possible that pass execution to the first sector of the first hard disk drive - the Master Boot Record (MBR) for boot instructions. The executable nature of the instructions in the MBR make them an attractive area for viruses (Skoudis & Zeltser, 2004). These viruses are called boot sector viruses. Other methods of infection involve overwriting legitimate programs as well as appending or prepending their instructions to legitimate programs (Skoudis & Zeltser, 2004).

Worms are similar to viruses with respect to their ability to self-replicate; however, they are standalone programs that do not leech on other programs before they can cause damage (Boyle & Panko, 2014). They are written to exploit vulnerabilities, using the Internet or the corporate Intranet as a medium of infection for effective and rapid propagation. They are composed of two components, the target selection algorithm and the payload. The target selection algorithm probes a host for its system attributes to determine if it is the intended or a susceptible victim platform and the payload contains code for the destructive tasks (Skoudis & Zeltser, 2004). Mobile code makes up the class of malicious programs written to take advantage of the ubiquity of the Internet and proliferation of applications on browsers and mobile devices. ActiveX plugins, Java Applets, JavaScript files are examples of propagation media for the spread of mobile code (Marek, 2002).

Trojans are malicious programs that appear benign and harmless. Spyware are a category of Trojans that mine and steal information from host system by logging keys, reading browser cookies, encryption keys and authentication parameters (Boyle & Panko, 2014). Trojans try to avoid detection by sometimes taking names of legitimate files and they form the basis of Advanced Persistent Threats (APTs). Remote Access Trojans (RATs) like Gh0st RAT and Poison Ivy give intruders remote control of victim computers (Daly, 2009). While Trojans can sometimes appear as system files to deceive users, they do not actually modify system binaries; however, rootkits on the other hand replace legitimate

system binaries. This is usually done to hide the presence of an intruder by modifying the output of legitimate system commands, files and libraries (Ligh, Adair, Hartstein, & Richard, 2010).

The free and open source nature of the Linux kernel has also made it attractive as the base operating system of a variety of platforms ranging from smart phones, servers, smart appliances and machinery giving rise to an increase in malware written for Linux (Damri & Vidyarthi, 2016). Ahnlab (2014) identified two methods of categorising current Linux malware – classification by malware purpose and by malware attack method.

The categories of malware when classified by purpose are exploits, Distributed Denial of Service attacks (DDoS), digital currency mining and backdoors. Exploits are written to take advantage of published and unpublished vulnerabilities to cause system instability. The availability of the source code also reduces the barrier to writing these exploits (AhnLab, 2014). Servers are attractive to malware authors because they contain valuable data which serves as an incentive for an intruder to insert backdoors to mine information from them. Servers are also expected to have high system uptime and high-end hardware specifications making them reliable hosts for launching malicious attacks against other systems in DDoS attacks as well as digital currency mining (AhnLab, 2014; Boyle & Panko, 2014). Rootkits are written to disguise the presence of other malware and the availability of source code makes it easy to write them (Messier, 2015).

2.3 Linux Operating System Internals

2.3.1 Internals

This section is a walk-through of the system organisation of a Linux based system. It undertakes a discussion of the operating system, its layers and the relationship and management of the underlying hardware. The internal operations, system services and calls are areas of the system where secure computing principles should be adhered to because vulnerabilities in these areas make malicious exploits possible (Bryant & O'Hallaron, 2015).

2.3.1.1 System Architecture

As illustrated in Figure 2.1, a Linux system is comprised of three layers – the hardware, operating system software and the user layers. The hardware layer consists of the Central Processing Unit(s), physical memory, disk drives(s), and Input/Output (I/O) peripherals. The operating system software is divided into two modes – the kernel space and the user space. The kernel sits atop the hardware layer and manages access to the hardware resources from user programs and processes as well as the CPU's access to memory and other peripherals. In kernel mode, a process has full access to all the system resources. The Linux kernel is described as monolithic because it has memory and file system management built into the kernel as against a microkernel architecture where those functions are implemented in user space with kernel responsible for coordinating messaging and signalling between processes (Tanenbaum & Bos, 2014).

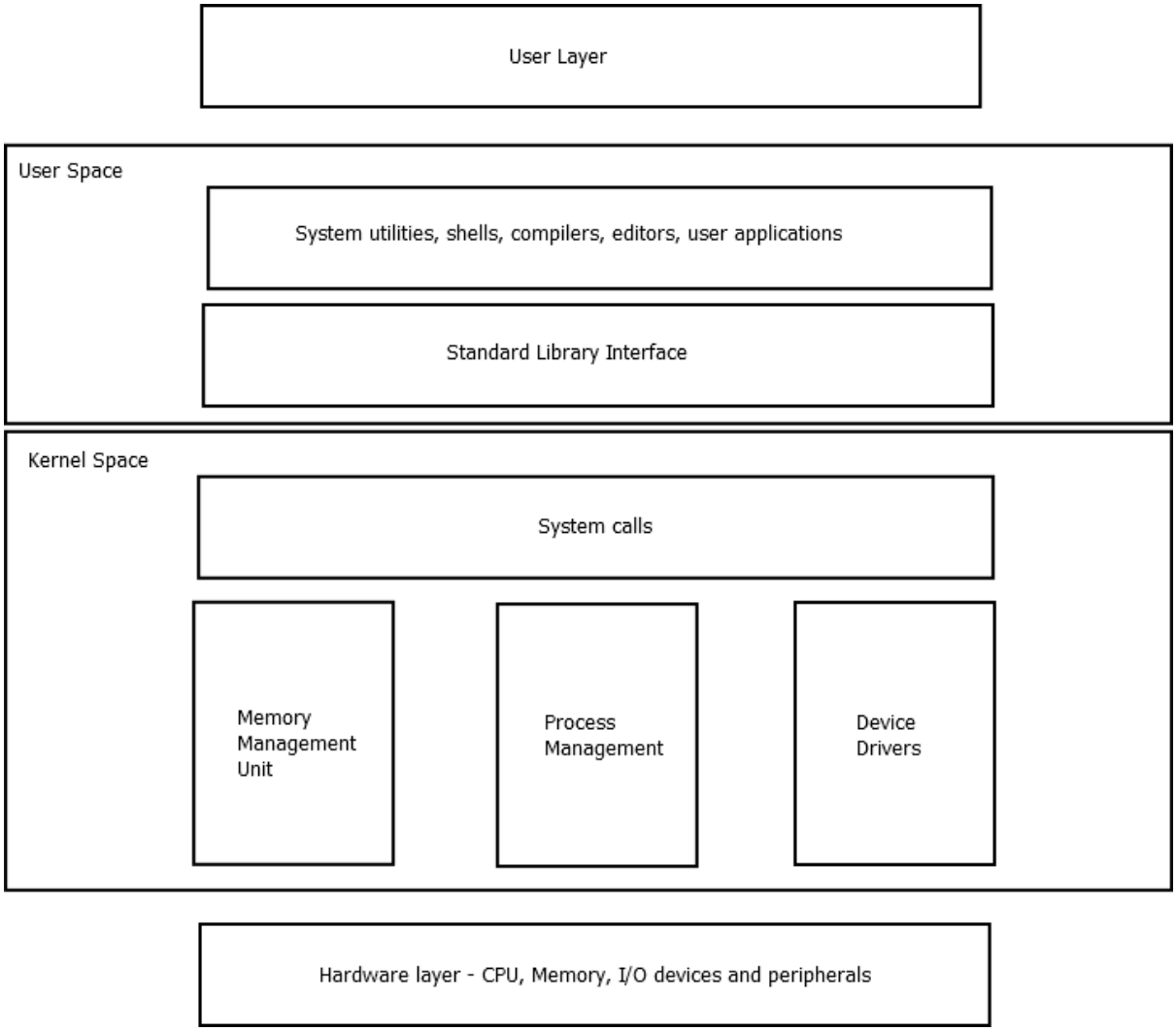


Figure 2. 1 System Organisation adapted from (Tanenbaum & Bos, 2014)

The kernel manages access to physical memory through the memory management unit and using the virtual memory construct enables the system to address a memory address space greater than physical memory (Tanenbaum & Bos, 2014). It provides mappings of virtual memory to physical memory addresses, maintaining a cache of recently used mappings and managing processes' page tables during context switching (Corbet, Rubini, & Kroah-Hartman, 2005).

User programs spawn processes and these processes require CPU resources and other devices to carry out their tasks. The kernel schedules access to resources from processes by managing communications between processes and enforcing a scheduling algorithm for access to CPU time while giving the illusion of simultaneous operation of the user programs (Ward, 2014). Processes are created, suspended, destroyed and execution mode changed from user to protected kernel mode through systems calls and signals. The process management subsystem also works with the memory management function to manage process access to the system memory and the handling of process signals indicating process states and requirements (Tanenbaum & Bos, 2014).

Love (2010) classified devices in Linux systems into character, block and network devices. Network devices are types of character devices because they share the attribute of only permitting sequential or stream based access to data. The data locations are generally not addressable or uniquely

identifiable (Tanenbaum & Bos, 2014). Other examples of character devices are printers, keyboard, and mouse. Block devices on the other hand allow random access to data because they have defined addressing (Love, 2010). Disk drives, physical memory are examples of block devices. The kernel manages these devices through device drivers and some are built into the kernel while others are loaded at runtime (Corbet et al., 2005).

The User layer access kernel managed resources using system calls implemented in the kernel. The user space provides standard library interfaces that are used by user programs and services. These library procedures are translated to system calls in the kernel and they are defined by the Institute of Electronic and Electrical Engineers (IEEE) under the Portable Operating System Interface (POSIX) 1003.1 standard to facilitate uniform APIs for portable programming on conformant systems. The user space of the Linux operating system also has application programs and utilities like editors, compilers, shells. They provide the means for user interaction and machine to machine communications, calling the standard library interfaces (which are mapped to system calls in kernel) when they need access to kernel managed resources (Tanenbaum & Bos, 2014).

The memory, process and device management functions of the kernel are addressed in more detail in the following sub-sections.

2.3.1.2 Memory Management

The memory management functions of the kernel are discussed in this section. The identification and instantiation of memory, allocation of pages and virtual memory management operations such as swapping and context switching are discussed in the sections below.

2.3.1.2.1 Pages, virtual memory and swap operations

Memory is an array of uniquely addressable bytes of storage. Pages are the smallest unit of memory allocation. They are typically 4KB on 32 bit architectures and 8KB on 64 bit architectures but these values can be configurable at kernel build time (Love, 2010). While a page refers to a unit of memory defined by the page size adopted by the operating system architecture, a page frame is a physical representation of pages on page sized aligned physical memory blocks. Pages are virtual or logical representations while page frames are concrete instantiations of physical memory and are uniquely identified in the kernel by Page Frame Numbers (PFN) (Tanenbaum & Bos, 2014).

The virtual memory feature of the Linux operating system lets each process run its own memory address space. This provides memory access control and protection, preventing one process from interfering with the address space of another and separating user mode and kernel mode processes. User mode and kernel mode programs refer to virtual memory while peripheral devices use physical memory addresses during Direct Memory Access (DMA) (Silberschatz, Galvin, & Gagne, 2013).

The Memory Management Unit (MMU) is a part of the CPU that manages the translation from virtual memory to physical memory. Complete mappings of virtual to physical addresses are held in page tables. The page tables are implemented in memory in a hierarchical arrangement for efficient use of system storage. They divide the virtual memory address space into sections with each section serving as an index to a table which either has an entry for another table lower in the hierarchical order or to the physical page itself. Each process has its set of page tables with the threads spawn from the same process sharing the same set of tables (Love, 2010; Tanenbaum & Bos, 2014).

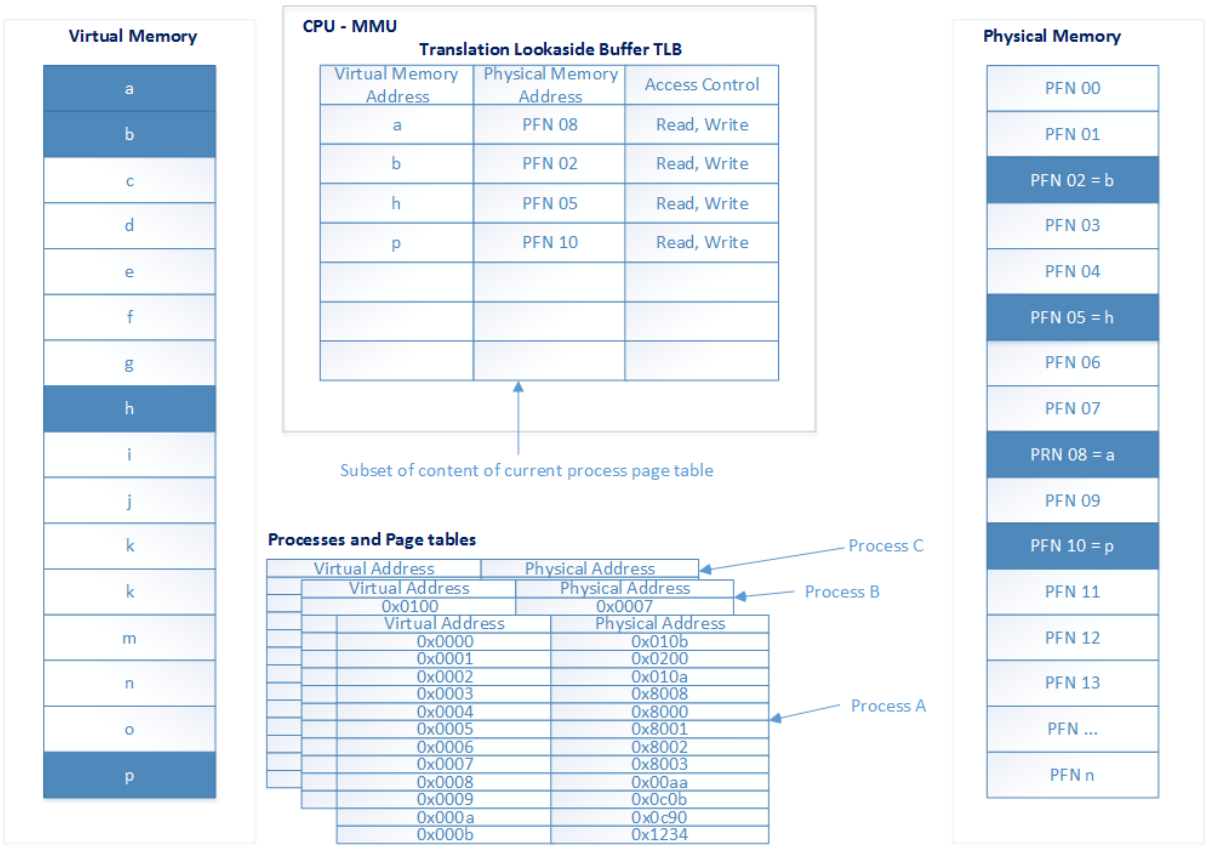


Figure 2. 2 Memory mapping operations adapted from (Tanenbaum & Bos, 2014)

In order to prevent two look-up operations for each memory access - the first one to determine the physical memory location of the required page table and the second one to query the page table for the physical address of the required virtual memory address - the CPU keeps a mapping of virtual to physical addresses for the process it is currently executing. It keeps this mapping in a cache called the Translation Look-aside Buffer (TLB). This is illustrated in figure 2.2. The TLB has entries for virtual address to physical address mappings and access controls. The use of the TLB speeds up physical memory address look up operations but the TLB is a limited resource and there are times when the process has more mappings than the TLB can hold. When a process attempts to access a virtual memory location for which there is no corresponding physical memory address mapping in the TLB, a page fault error is thrown (Corbet et al., 2005; Tanenbaum & Bos, 2014). This triggers the page fault handler in the kernel which loads the required page table. This also occurs during process context switching. When CPU switches execution to a process and the process' page tables have not been loaded, the first virtual memory access by the process will result in a page fault which will cause the appropriate page tables to be loaded (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

Lazy allocation and memory swapping also result in page faults. During lazy allocation or demand paging, the kernel reserves pages immediately when they are requested by the user space application using *malloc* library call (Bovet & Cesati, 2005). Actual allocation of page frames only occurs at runtime when the running process needs access to the allocated pages. This is meant to be a performance and resource optimisation feature that ensures that if a reservation is not needed at runtime (not accessed by its process), it is not holding up a resource that should be available for other tasks. Typically, when

a request for memory is made, the kernel takes note of the request in its page table and it returns execution back to the user mode without updating the TLB. When the virtual memory addressed is referenced by the requesting process, the address returned to user is not seen in the TLB so a page fault exception is generated which returns control to the kernel (through the page fault handler) (Tanenbaum & Bos, 2014). The kernel, seeing that the allocation is valid, allocates page frames in physical memory for the virtual address and records the address mapping entry in the TLB before returning execution to the user mode application. For applications that want to avoid the performance penalty introduced by this feature, the library function call *memset* can be used to initialise the allocation or the use of *calloc* instead of *malloc* in the memory request (Kerrisk, 2010; Silberschatz et al., 2013).

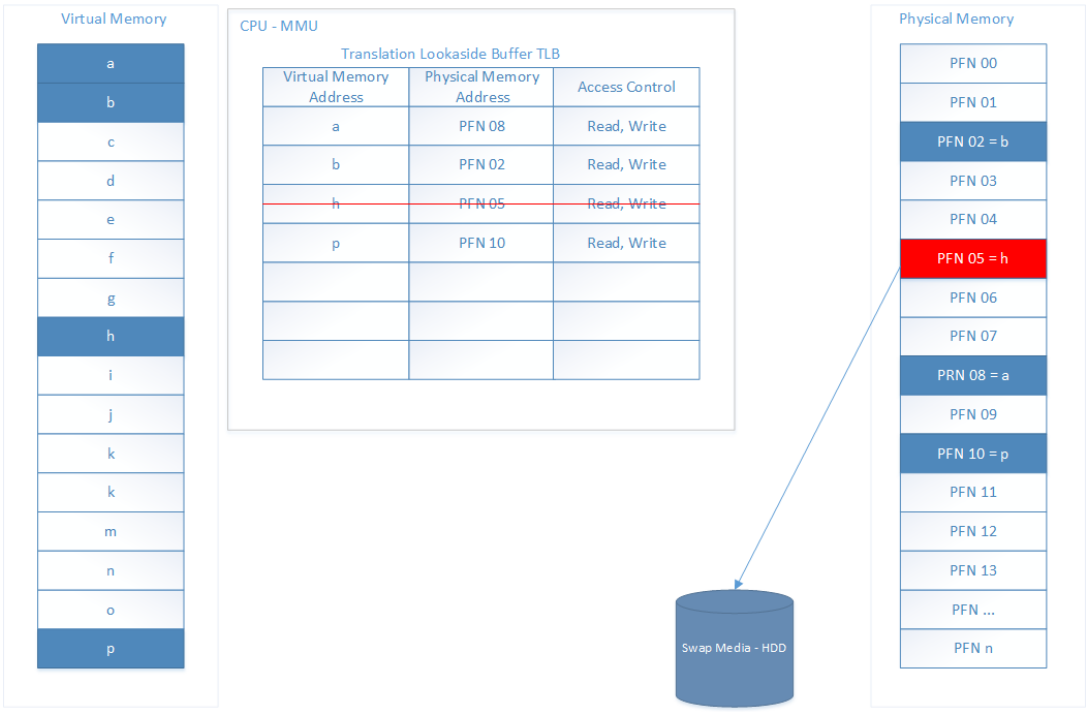


Figure 2. 3 Memory Swap-out Operation adapted from (Bovet & Cesati, 2005)

Swap operations as shown in figures 2.3 and 2.4, are done to free up space in the physical memory. Swapping allows the total memory allocated to be greater than the physical memory installed. It involves moving pages from the RAM to the disk and marking the entry in memory tables as swapped. A page is swapped out when it is copied from the physical memory to the swap media (hard disk) and its entry in the TLB removed. There is still a reference in the user mode process pointing to this frame as it is still in the user virtual address space for the process. When it is needed, a page fault is generated because the entry is not in the TLB. The page is copied back from the swap media back to the physical memory (swapped in) and its entry updated in TLB. The library function calls *mlock* and *mlockall* can be used to prevent an allocation from being a candidate for swap out (Bovet & Cesati, 2005; Tanenbaum & Bos, 2014).

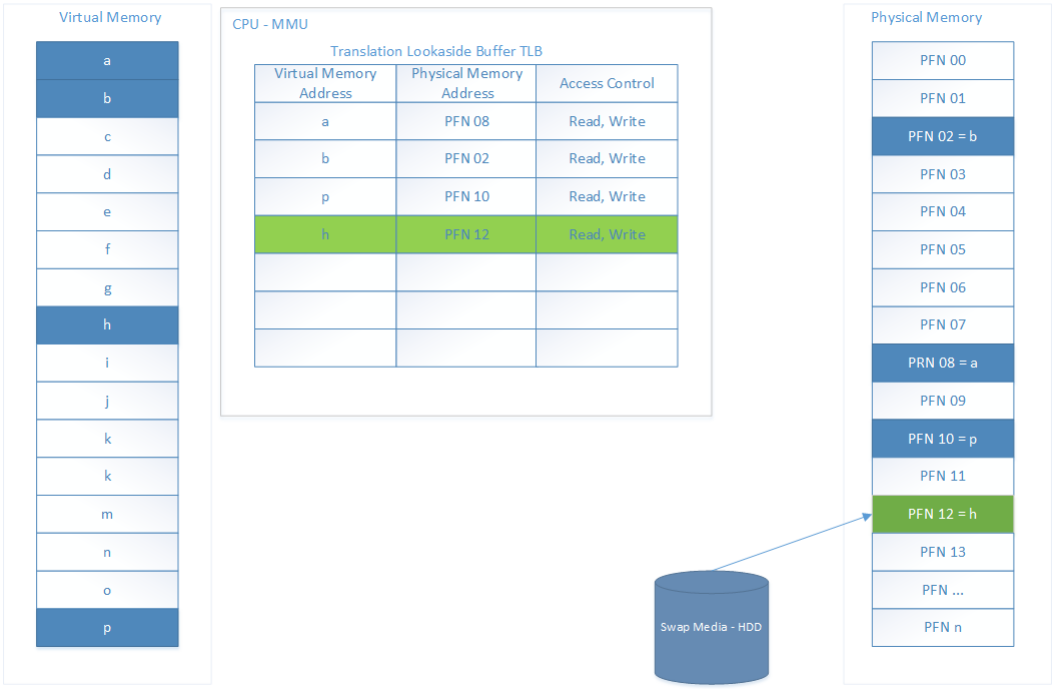


Figure 2. 4 Memory Swap-in Operation adapted from (Bovet & Cesati, 2005)

2.3.1.2.2 Memory Zones

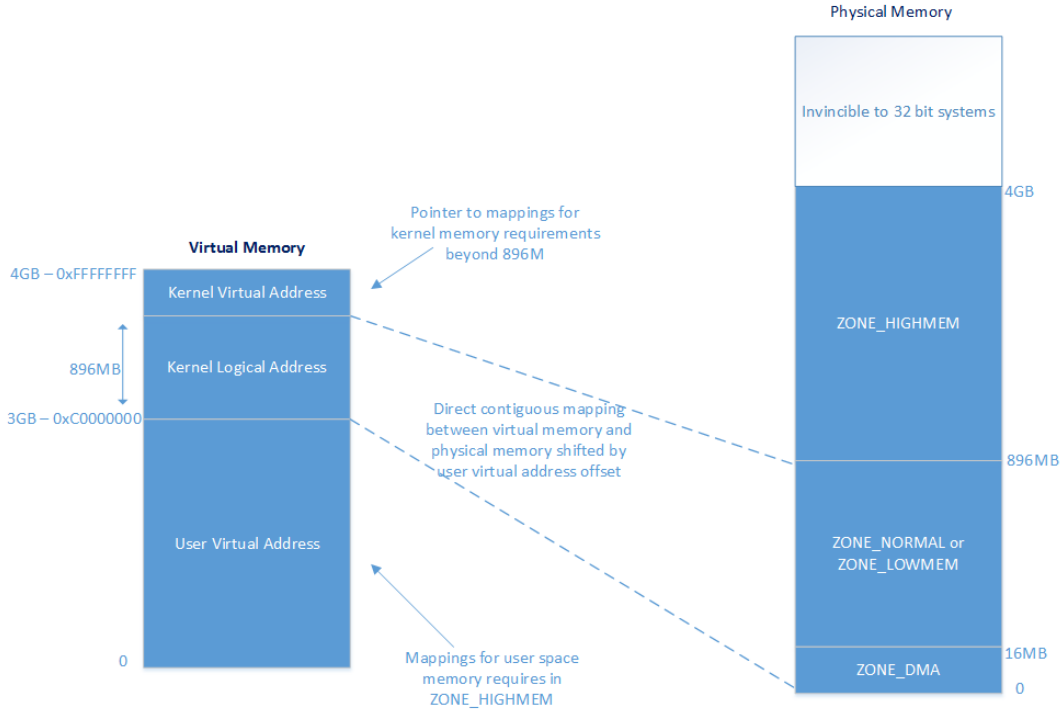


Figure 2. 5 32-bit System Memory Zone adapted from (Corbet et al., 2005)

The physical memory can be divided into zones and these zones and their allocations are dependent on the memory systems in use by the CPU and the I/O hardware. The zones are ZONE_DMA, ZONE_DMA32, LOW_MEM and HIGH_MEM. *ZONE_DMA* is the lower 16MB of physical memory that is addressable by some old hardware devices such as Industry Standard Architecture (ISA) devices.

These devices can only access the lower 16MB of physical memory using direct memory access. *ZONE_DMA32* is the 32-bit address visible to other devices capable of direct memory access but are however limited by being 32-bit devices which have a memory addressable region of about 4GB (Love, 2010). Figure 2.5 illustrates the virtual memory to physical memory zone mappings for the 32-bit systems.

On 32-bit architectures, the first 896MB area that is directly addressable by the kernel is referred to as the LOW_MEM or Normal Zone. This region is used for kernel logical addressing. The *kernel logical address* is a predictable one to one mapping to the physical address, usually by a fixed offset (the size of the user virtual address). It can be used for contiguous memory allocations (Corbet et al., 2005). This memory region has a direct one to one mapping to physical memory and can never be swapped out or paged out. Contiguous allocations made in this area are always contiguous in the physical memory making them suitable for operations and processes requiring direct memory access. The *kmalloc* and *kfree* library functions are used for requesting and deallocating the kernel memory in the LOW_MEM zone (Love, 2010). *kmalloc* does a contiguous allocation of the physical memory. It accepts the required size as argument and returns a pointer to the address of the first memory location assigned if the call succeeded. Typically, more bytes than that requested are allocated because the kernel memory allocations are done in multiples of pages necessitating rounding to the nearest page boundary (Kerrisk, 2010).

The High memory zone (HIGH_MEM) is typically applicable on 32-bit architectures where a total of 4GB of physical RAM is supported. In 32-bit systems, the kernel can only directly address the first 896MB of the first 1GB. The 1GB - 4GB address region that the kernel is unable to logically map to is the HIGH_MEM zone. It is used for the virtual memory allocation of the kernel and user space processes. On 64-bit systems, the kernel has access to the full memory address range so all memory is LOW or normal memory with no concept of high memory zone (Corbet et al., 2005). This is illustrated in Figure 2.6.

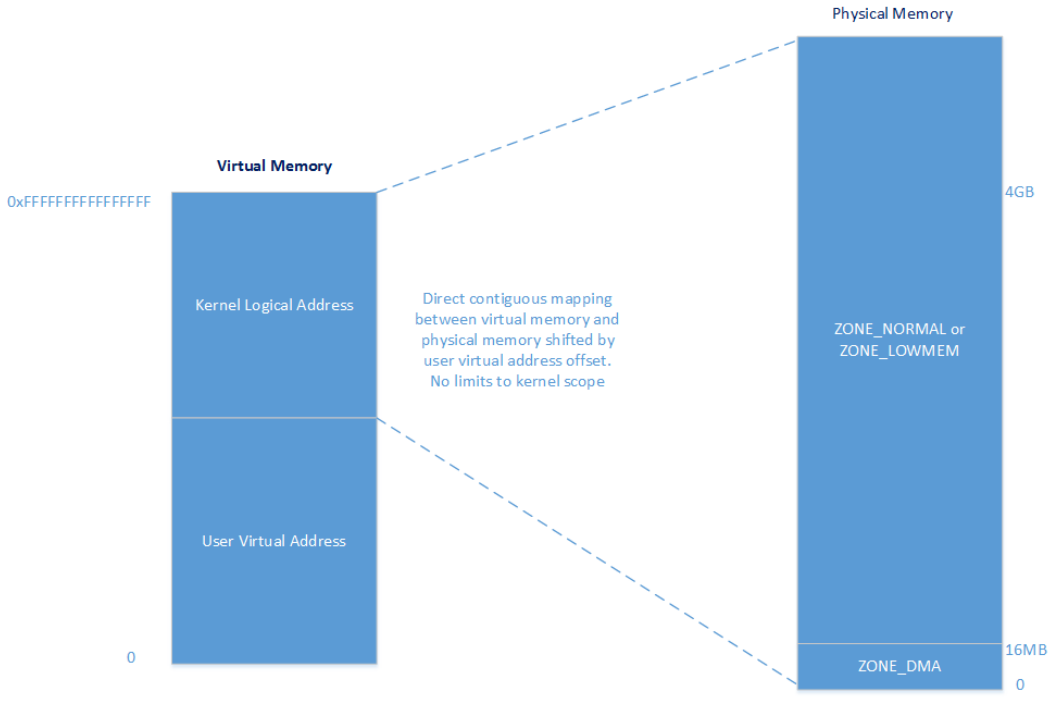


Figure 2. 6 64-bit System Memory Zone adapted from (Corbet et al., 2005)

The 104MB area at the top of the kernel virtual space (between 896MB and 1024MB) is reserved for non-contiguous allocations in 32-bit systems for the physical memory requirements of the kernel beyond the 896MB point. The kernel creates mappings in the 897MB – 1024MB region for allocations it has made in the physical memory beyond the 896MB area. The *kernel virtual address space* does not have the one to one mapping with the physical address space. It is used as a pointer to reference the memory addresses in areas beyond the kernel logical addresses and the non-contiguous memory mappings that might require large buffers (Corbet et al., 2005).

Bulk memory allocations and deallocations beyond the LOW_MEM zone by the kernel for the kernel space processes are with the *vmalloc* and *vfree* library functions. The difference between *kmalloc* and *vmalloc* is that the allocations made with *vmalloc* can be logically contiguous but they are not physically contiguous, requiring mapping entries for the translation between the virtual address space to the physical address space maintained on the page table for the process. In contrast, *kmalloc* allocations that are logically contiguous are also physically contiguous with a fixed offset as the translation value (Corbet et al., 2005). Allocations by *vmalloc* use more entries in the TLB because of the distributed and dispersed allocation as each page in the virtual memory must be mapped to its corresponding location in the physical memory. User space programs use the *user virtual address* and its size is architecture dependent and forms the fixed offset for the translation of the kernel logical addresses to physical memory addresses (Bovet & Cesati, 2005). User space processes use the *malloc/calloc* library calls for allocation. The difference between them is that malloc uses a lazy allocation while calloc initialises the requested allocation to zero. Both library calls use *mmap* and *brk/sbrk* system calls in the background. The former is for large allocations while the latter is for incremental allocations such as for the increase of memory heap size (Kerrisk, 2010).

2.3.1.2.3 Zone Allocators

Each memory zone has a zone allocator that addresses the efficient allocation of pages in physical memory reducing fragmentation. Memory allocations for user mode processes are generally done with the *buddy* system of allocation. The buddy system of allocation involves neighbouring memory blocks combining or a memory block splitting by a power of two. Information is maintained about used and unallocated blocks in memory using kernel data structures. To reduce occurrences of external fragmentation, which occurs when a process requires a large allocation but there is no single block to handle the request necessitating the need to split the allocation to different areas, this allocation method allows unused adjacent blocks of same sizes to be combined into a buddy heap. This buddy heap can also combine recursively with an adjacent block of same size that is unused. This process allows the allocator to handle large requests. For smaller requests, a block recursively splits by a power of two in size (its square root) until it gets to a defined lower limit or the smallest size that can accommodate the request with least amount of waste (internal fragmentation). This lower limit is usually the page size of the platform. The choice of the lower limit involves trade-offs. A very low value reduces memory waste (internal fragmentation) but has more overhead as there are now more blocks to track with the kernel data structures (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

The kernel allocations for data structures and kernel objects like *inodes*, *task_struct* objects change frequently and require as little fragmentation as possible because the kernel allocations have direct mappings to physical memory. These foregoing factors necessitate the use of *slab* allocation method to increase the speed of allocation and reduce fragmentation. In this method, the system memory is

partitioned into physically contiguous areas (using the buddy system) called slabs with each variable sized slab dedicated to different types of kernel data structures/objects. Each kernel object has a cache that is made up of one or more slabs. These caches maintain pointers to empty, partially used and full slabs for each type of data structure. When a request is made, the data structure is allocated to a free object in a partially free slab for that type data structure involved thus reducing internal fragmentation. When memory is freed, the kernel simply marks the object as unused in the cache, freeing it up for subsequent allocations (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

2.3.1.3 Process control

The process management function of the kernel is discussed in the following sub-sections. The system calls and signals for process (and thread) management and communication are briefly described as well as kernel synchronisation and task scheduling.

2.3.1.3.1 System calls and signals

Each process in Linux has a unique Process Identifier (PID). Process creation in Linux is achieved with the *fork* system call. Historically, this call copied the memory image and variables related to the parent (creating or calling) process into the child (created or called process) allowing the child to have access to the files opened by the parent process (Tanenbaum & Bos, 2014). The *fork* system call causes the kernel to return twice and the value of the return values are the PID of the child when it is returning to the parent and PID 0 when it is returning to the child (Love, 2010). The g*etpid* system call is used by the child to get the PID of the parent if it needs it. This is important because as children spawn more children processes, a complex family tree is easily being formed (Kerrisk, 2010).

Shell commands launched by processes are created using the *exec* system call. This call replaces the memory image and environment variables of the parent process with those of the commands invoked. The w*aitpid* system call is used when a parent needs to wait on a child process and the parameters for this system call are the child PID (or any child denoted by PID of -1), address of the variable holding the value of the exit status and a parameter to determine if the call should be blocking or return if no child has been terminated yet. A completed child process without a parent process to return control to is a zombie process (Love, 2010).

The uniqueness of PIDs allow for the communication of signals and messages among processes. A collection of processes in the same family tree (a process group) can send signals to each other. These signals can be instructions to restart a process or to instruct it to re-read its configuration file *SIGHUP (*terminate gracefully), *SIGTERM* (terminate unconditionally and immediately), *SIGKILL* or SIGILL (suspend itself) (Shotts Jr, 2012). A process implements the *SIGACTION* system call to determine how it wants to handle a signal (included as one of the parameters). If *SIGACTION* is implemented, on receipt of the signal referenced in the *SIGACTION implementation*, control passes to the handler. SIGKILL goes straight to the kernel and is never handled by *SIGACTION (Tanenbaum & Bos, 2014)*.

2.3.1.3.2 Process data structure

Each process has a user part (with associated program counter and memory stack) and a kernel mode part when one of its threads makes a system call giving it access to the machine resources with its own kernel mode stack and program counter. Processes and threads in Linux are represented internally as tasks with a *task_struct* data structure (Bovet & Cesati, 2005). Each user level thread has

in the kernel a task structure and for each process, there is a process descriptor of *task_struct* in memory with information for the management of all the processes, open files and scheduling parameters. Internally, the kernel organizes all processes in a doubly linked list of task structures with PID as keys to the address of the *task_struct*. Some of the variables held or referenced by the process descriptor are the scheduling parameters, memory image, signals, machine registers, system call state, file descriptor table, kernel stack (Tanenbaum & Bos, 2014).

2.3.1.3.3 Copy on write

During the *fork* operation, the operating system does a *copy on write* to conserve the system memory. The expectation is that the parent memory stack and its other resources should be copied to the child so that both can work without writing into each other's space but in modern implementations of the Linux kernel, each child has separate page tables but point to the parent's table (Love, 2010). If either the parent or child subsequently need to write to that page, a page protection fault exception is thrown and a copy of the page is created which they can then write to it, hence *copy on write*. This form of demand paging is done to reduce memory requirements and overhead in process creation. It also turns outs to be efficient because in a lot of cases, the children processes might not need to refer to the parent process resources because they (parent) are either terminated shortly after being spawned or there is a need to call the *exec* system call to launch another program whose pages and memory image replaces theirs (Love, 2010).

2.3.1.3.4 Processes, threads, tasks, clone system call and scheduling

Historically, all processes spawn threads and all threads share file descriptors, signal handlers, address space, alarms and other global properties while maintaining unique registers, but modern Linux kernels introduced the concept of these parameters being thread or process specific with the introduction of the *clone* system call (Tanenbaum & Bos, 2014). When a process is created with the *clone* system call, if it shares nothing with its parent, it is given a unique PID but if it does, it is given the same PID but a different Task Identifier (TID) and both fields are stored in the *task_struct (Tanenbaum & Bos, 2014)*. The clone system call has bit map parameters to define the resource sharing mode (Kerrisk, 2010).

The kernel deals with user modes processes which are kernel mode processes that are a consequence of user mode processes making systems calls as well as internal kernel code operations called by I/O devices. There are two algorithms for managing process access to the CPU resources. The first is the *Completely Fair Scheduler (CFS)* which is for processes that are non-real-time time sharing processes. CFS uses two configuration parameters - the minimum granularity and the target latency. The latter is the time interval within which every runnable process should have run once. Basically, all runnable processes using the niceness value as weight are given proportional access to the CPU. if all processes have same weight, they will be able to run (1/N) * the target latency where N is number of runnable processes. Those with higher priorities based on niceness value are run more often and those with lower priorities (higher niceness values) are run less frequently. The minimum granularity addresses the inefficiency of context and processor switching when there are many runnable processes. It is the minimum amount of time a process has access to the processor. This reduces costs associated with switching between processes (Tanenbaum & Bos, 2014).

For real-time scheduling, there are two types of such processes. Real Time First in First out (FIFO) and Real Time Round-Robin. Both run and are pre-empted by processes with higher priorities or if priorities

are equal, processes that have been in the wait state longest are prioritised. The difference is that the FIFO process is not interrupted periodically but will run until it blocks or exits (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

2.3.1.3.5 Kernel Synchronisation

When kernel code is running, it is usually using some internal data structure. To maintain the integrity and consistency of these data structures, there needs to be a scheme to protect the kernel processes from interfering with each other and yet operate efficiently because if a process is not allowed to be interrupted to prevent corruption of the data structures it is working with, the system operation can degrade because there might be I/O devices waiting around and unable to run their processes. To balance these conflicting goals, interrupt processes are categorised into four (4) classes. The classes in increasing priorities are User mode programs, kernel service routines, bottom-half interrupt service handlers and top-half interrupt service handlers. Only the user mode programs can be pre-empted by processes in similar categories, others can only be pre-empted by other processes in higher categories (Tanenbaum & Bos, 2014).

2.3.1.4 Device Control

Device and I/O management in Linux is done through the device drivers (Tanenbaum & Bos, 2014). The management of access to these resources from processes is a function of device management. The file centric nature of the operating system is also examined below in the discussion of the virtual file system and the organisation and features of the supported file systems.

2.3.1.4.1 Device Drivers

Device drivers form the basis of I/O operations in the Linux operating system. One device driver usually handles I/O for a device type. This relationship is defined in the kernel with major and minor device numbers. All devices of same type (either block or character devices) with the same major number generally use the same device driver. The minor number comes to play when a device driver needs to differentiate between different instances of the same device it controls. The kernel has internal hash tables of data structures for character and block devices. These objects are pointers to the procedures for the functionality supported on a device. When a user access one of special files representing a device, the filesystem determines the minor and major number and selects one of the kernel hash tables depending on if the device is a block or a character device. I/O devices are integrated into the file system and accessed as special files in the /dev/ directory. These special files are broadly divided into two categories - block and character files. There are two parts to a device driver and while both parts exist in the kernel, one part is the interface to the user process while the other part interacts with the device. The drivers enable direct interaction with the kernel, calling procedures for memory allocation, DMA control, timer management etc. (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

The block device subsystem maintains the performance for disk devices by the scheduling of I/O operations. The request manager manages the read and write operations with buffering as an intermediate operation. Linux used the Completely Fair Queuing (CFQ) I/O scheduler for handling I/O operations. It maintains a set of lists - one for each process so every request from a process goes into the list maintained for the process. A specified number of requests are withdrawn from each list at each I/O operations interval (Tanenbaum & Bos, 2014).

Character device drivers when registering must also notify the Linux kernel of the set of functions (I/O operations) that they implement. The drivers implement the line discipline that dictates the formatting and encoding of data stream either controlling terminal input and output or network protocols like PPP and SLIP (Tanenbaum & Bos, 2014). The Linux kernel networking subsystem is implemented in three modules - the socket interface, protocol drivers and network device drivers. The socket interface is used by user applications to perform all network related operations. It provides an abstraction for the possibly wide range of networking protocols supported by the kernel isolating the user applications from the complexity. The protocol stack contains the set of procedures by which the devices will communicate. The functions at this layer include error checking and reporting, packet sequencing and fragmentation, reliable transfer and routing. Device drivers are the abstractions of the networking device hardware for remote communication (Silberschatz et al., 2013).

2.3.1.4.2 Virtual File System

Linux conceptually treats everything that can take input and provide output as a file. These include conventional directories and files, network connections, device drivers. It provides as abstraction for file operations using the Virtual File System (VFS). VFS defines file system objects and the operations that can be performed on them enabling the kernel to perform the equivalent operation on a specific object while on the higher layer, programmers and users work with generic library and system calls (Tanenbaum & Bos, 2014).

The file system objects are the inode, file descriptor, superblock and dentry objects. The inode is a representation of an individual file, it is a pointer to the data block belonging to a specific file. Each file, directory, network socket is represented by a unique inode object. The file descriptor object represents an open file. Each process has a file descriptor object for each file it opens. This object tracks the state of the file, the access requested, when it was opened and its modifications. It is possible for multiple processes to open a file. The inode is same for the file but the file descriptor is unique per process allowing for simultaneous alteration of files. The superblock object provides access to the files represented as inode to processes. There is a superblock object for each disk and network file system mounted. Every inode is uniquely identified by a unique file-system/inode number pair. A dentry object is a directory entry and it includes the directory and file name in the path name of a file. When a file is requested, the inode for each folder(dentry) in the directory tree is resolved until the file itself is reached. A dentry cache is kept for each file name translation to speed up subsequent requests for files or folders (Silberschatz et al., 2013; Tanenbaum & Bos, 2014).

The file system objects are implemented on physical media as file systems which determine their arrangement and access by the kernel. Ext3 is the most popular file system used on Linux. It added journaling support to the file system. This is the use of a separate dedicated area on the disk for storing file system changes, operations and metadata. This reduces the possibility of file system corruption during system crashes. Ext3 supports a maximum individual file size of between 16GB to 2TB and an overall file system size of between 2TB to 32TB. It stores files on the disk in block sizes of 1,2,4 or 8KB depending on the architecture. Ext3 supports a maximum of 32,000 subdirectories in a directory. Its allocation policy seeks to ensure that during I/O operations, several disk blocks can be read in a single operation instead of reading at single block sizes. It does this by partitioning the file system into block groups, allocating files in same block groups as their inodes and the inodes themselves are stored in same block group as their parent directories for non-directory files. Directory files are kept in different block groups where possible. These policies ensured that related information are kept together and

disk contents are spread around disk groups and subsequently across the disk (Silberschatz et al., 2013).

Ext4 is the latest iteration of the Linux file system. It has some advantages over ext3. It supports a maximum individual size ranging from 16GB to 16TB and an overall maximum file system size of 1 EB. Directories can contain more than 64,000 subdirectories and there is an option to turn off journaling as the overhead might not be required on simple systems. Before Ext4, these systems typically used the older ext2 file system to avoid journaling (Linux Kernel Organization, 2016; Tanenbaum & Bos, 2014).

Extents are the significant new addition to ext4. Ext3 keeps track of allocations (file data to block group) by maintaining a bitmap of free blocks in a block group. Journaling is also done at block level. This can be inefficient for bigger files hence the use of extents by ext4. An extent is contiguous sequence of blocks that indicate the data in a file. Instead of tracking a pointer to each data block that a file inhabits, extents keep a pointer to the start and end or size of the data block consumed by file. Entries are kept for extents and multiple extents can represent a file if a contiguous allocation of blocks is not available. Tracking at extent level often means that there is less overhead for journaling as extents for a file are monitored not each data block of the file. Ext4 is the default file system in new installations for most Linux operating system distributions (Linux Kernel Organization, 2016).

The process file system is a special file system that does not store data permanently but instead changes its contents stored in the /proc directory dynamically based on the state (command line, environment variables, signals, masks) of the processes running on the system. Each running process has a directory in /proc and the file contents are a representation of the process states. Inode numbers are 32 bits long and PIDs are 16 bits in size. The first 16 bits of the inode of the files in the /proc directory are the 16 bits of the PID of the process while the remaining 16 bits define other information about the process. As PID 0 does not exist, the inodes with the PID field value of zero report global information about the system such as the kernel version and operating statistics (free memory, CPU load, I/O utilisation, device drivers running etc). The /proc/sys directory provides access to the kernel variables and values in American Standard Code for Information Interchange (ASCII) decimal can be read and written to these variables. The system call *sysctl* can be used to edit these values by passing binary numbers to set and unset a parameter (Tanenbaum & Bos, 2014).

2.3.2 Forensic Artefacts

This section is a review of parts of the Linux system that can be checked for indicators of compromise when the presence of malware is suspected. The log files, the binary file structure and memory and configuration files are discussed below.

2.3.2.1 Log files

Log files are a record of the activities on the system by users and processes. System services and daemons have their respective log files to report on the activities of the services as well as errors if there are any. Log files are a good indication of the state of the system and are usually modified by attackers to hide their activities whenever possible. Log file sizes and archive settings indicate how long the system keeps a process' log files. With respect to system integrity, log files can be divided

into user activity logs, and system logs comprising of application and process files (Malin, Casey, & Aquilina, 2008).

The user activity logs are files that hold user related information on the system. Examples are /var/run/utmp, /var/log/wtmp, /var/log/lastlog, /var/log/btmp files that store information about current logged on users, historical logons within a log rotation period, last time all users logged on to the system and all failed logon attempts respectively. These files are binary, not world-readable files. They are less prone to being altered to conceal a specific activity. They can be deleted with the appropriate privileges and the executable files that generated them can be altered to give false outputs; however, these would serve as definite signs of intrusion. Every user has a command history file listing all commands entered on the shell by the user. This file is text editor modifiable with the right access and it does not have time stamp information (Malin et al., 2008; Nelson, Phillips, & Steuart, 2014).

Application and process logs are logging files in the /var/log file directory tree. These are system log files that give information about specific processes. A web server like Apache for instance logs information about its operations, the requests it is handling and any errors either system or user related ones that it encounters. Depending on distribution, these files can be stored in the /var/log/apache2 or /var/log/httpd directories or any other location determined by the administrator. Each log entry is time stamped and it indicates the facility (the part of the Linux system the message concerns for example *auth* and *kern* represent for authentication and kernel related logs respectively) that generated it, the severity, a descriptive message among other bits of text. These files are plain text also and they can be manipulated with the right access permissions (Malin et al., 2008).

2.3.2.2 Memory

Memory images of a live system hold information about the activities of system users and processes as well as malware events. Open network sockets, encryption keys, time stamps of system calls are some of the information available in memory images that are useful during investigation of malicious activities on a system (Malin, Casey, & Aquilina, 2013).

Memory forensics involve acquisition and analysis. (Ligh, Case, Levy, & Walters, 2014) identified three historical schemes for memory acquisition on Linux based systems - the /dev/mem and /dev/kmem device files and the *ptrace (Linux Programmer's Manual, 2016)* system call. These methods were only viable on 32-bit systems. The */dev/mem* device was made available to tools like *dd* and *cat* for reading and writing. It allowed export of physical memory which potentially contained protected areas, device memory and the unmapped physical addresses of memory presenting the risk of memory corruption. Its acquisition capability was limited to the LOW_MEM region (896 MB). */dev/kmem* exported kernel virtual address space with the attendant risk of exposing kernel space to user space applications, giving rise to the possibility of memory corruption. The risks to system stability presented by */dev/mem* and */dev/kmem* is the reason for disabling them by default on most recent distributions. The *ptrace* system call dumps the memory image of the process under investigation not the complete memory footprint. It gets the process memory footprint from */proc/<pid>/maps* file and dumps the pages to disk. Forensically, It is only suitable in cases where a process memory footprint is under investigation (M. H. Ligh et al., 2014).

Fmem is a kernel module that creates a character based device driver /dev/fmem when loaded. This presents a view of physical memory by reading the /proc/iomem file to get the memory allocation

layout so that investigator can avoid device memory and unmapped addresses. Linux Memory extractor, *LiME (504ensicslabs, 2017)* is another kernel module but it does not create a user mode device driver interface thereby improving accuracy by avoiding user space and kernel space interactions and context switch during acquisition (Ligh et al., 2014).

The use of the /proc/kcore file avoids the need to have a loadable kernel module for memory acquisition. The kernel keeps a mapping of its virtual memory address space in the /proc/kcore file. On 64-bit systems, since all physical memory is in kernel virtual memory, the complete picture of the physical memory can be gleaned by exporting the /proc/kcore file. There are limitations with this method on 32-bit systems because on such systems only the first 896 MB of physical memory is mapped to the kernel virtual memory. Linux systems have different memory map and data layout structure depending on kernel version hence the need for loading specially compiled kernel modules for the target system when using the *Fmem* and *LiME* kernel drivers for memory acquisition. The use of the /proc/kcore file for acquisition while avoiding this compilation, merely moves the requirement of getting the kernel profile to the analysis phase from the acquisition phase. The analysis of kernel memory dump acquired using the export of /proc/kcore file involves getting a suitable kernel memory map profile of the target system based on the Linux kernel version in use (Case & Richard, 2017). The /proc/kcore file needs to be enabled in the kernel for this acquisition to be possible; however, this is enabled by default on most stock Linux operating system distributions (Ligh et al., 2014).

2.3.2.3 Executable and Linkable Format

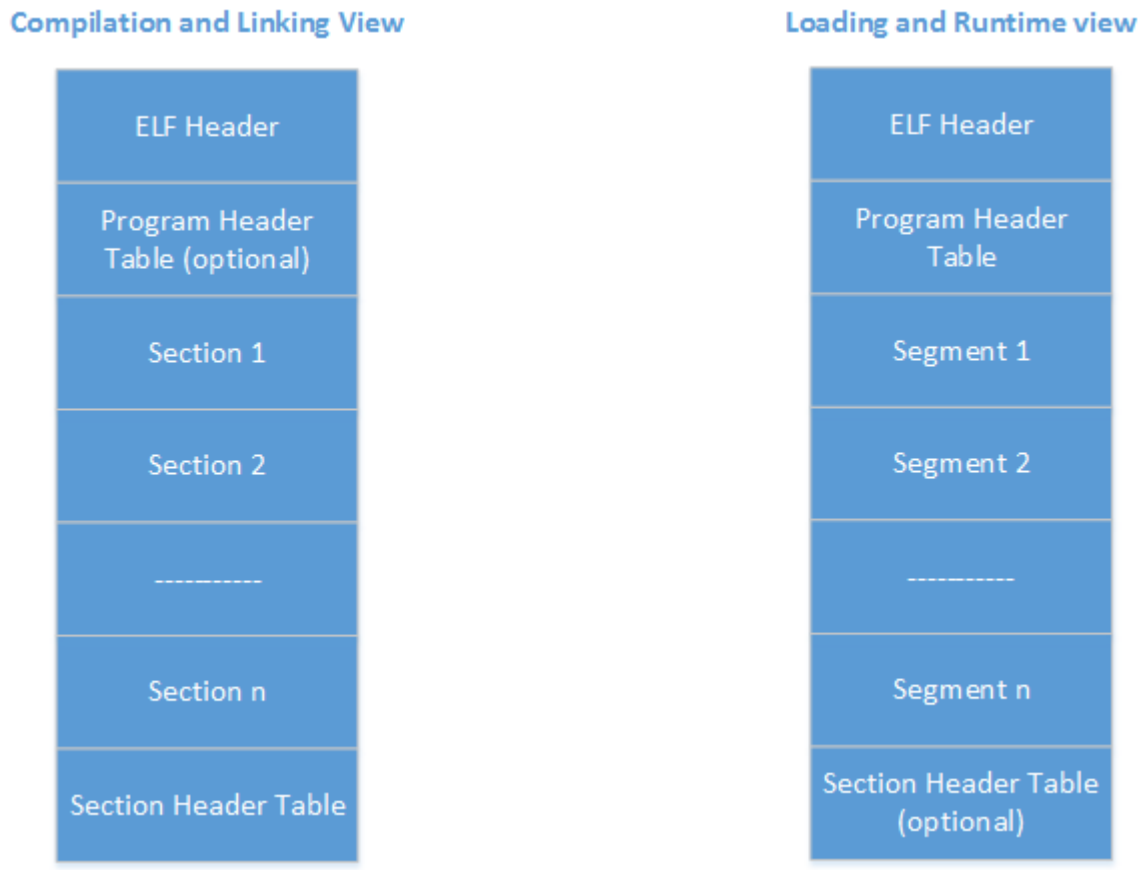


Figure 2. 7 ELF File Structure adapted from (Tool Interface Standards Committee, 2001)

Executable and Linkable Format (ELF) is the executable format of the Linux file system analogous to Windows executable (exe) and dynamic link library (dll) files. It is the format for user applications, shared libraries, core dump files, kernel modules and the kernel itself. An object using ELF format consists of an ELF header followed by program header table and or section header table or both. The ELF header is always at offset zero of the file. The section and program header table's offset are defined in the ELF header. They describe specifics of the file. The ELF header has fields indicating the magic number or file signature of the executable *e_ident*. This can be searched in the memory dump to indicate the beginning of executable program execution, processor architecture and word format big or little endian. The *e_type* field indicates whether the file is a relocatable, executable, shared object or core file (Tool Interface Standards Committee, 2001).

The ELF file consists of sections and segments which are indexed by Section Header Table (SHT) and Program Header Table (PHT) respectively. PHTs are important for executable and shared object files as they describe how a process image that is loaded onto the system memory should be built. They are important at program runtime. This is shown in the loading and runtime view of Figure 2.7. Sections hold information about program linking - instructions, data, symbol table and relocation information so they are important to relocatable files. They are important at compile time and files used during program linking must have SHT as illustrated in the compilation and linking view of Figure 2.7. Shared libraries are functions and variables dynamically loaded to other executables or shared libraries. They are typically stored on disk as *.so* files. They are primarily used so that the need to create static links in programs is avoided, thereby reducing the memory size of a program (Tool Interface Standards Committee, 2001).

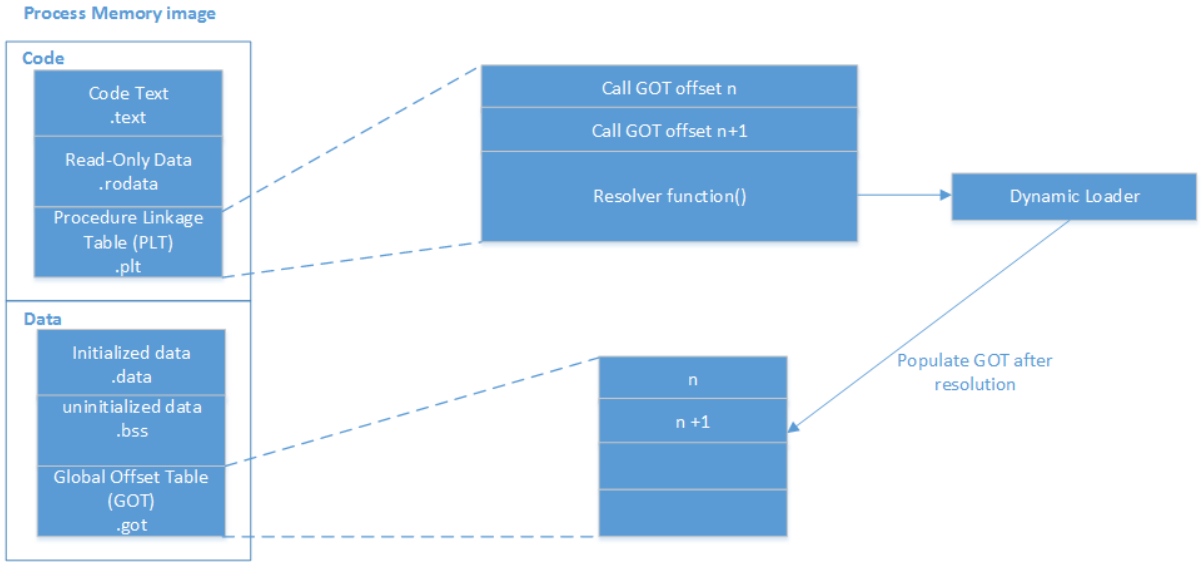


Figure 2. 8 Function call address resolution adapted from (M. H. Ligh et al., 2014)

The concept of shared libraries can be exploited by attackers to cause harm. This is because when a program is created as position independent, it cannot have absolute virtual addresses for the global variables and shared library functions it uses. The global variables are referenced as an offset to the ELF headers in the Global Offsets Table (GOT). GOT is a list of symbol addresses that cannot be computed at runtime. A program references the GOT at runtime to resolve the virtual address for the global variable. The Procedure Linkage Table (PLT) is used to resolve the address of functions from shared libraries that cannot be resolved at runtime. Each entry in the PLT is a reference to an offset in

the GOT which specifies the virtual address to the function call. If the function call has not been called before, the GOT's entry will point back to the resolver routine in PLT which is a call for the dynamic loader to locate the virtual address of the function call which is then populated in the GOT for subsequent calls. The foregoing is illustrated in figure 2.8. Attackers can manipulate the entries in GOT thereby forwarding library functions calls to malicious code (Ligh et al., 2014) (Malin et al., 2013).

2.3.2.4 Configuration files

The configuration files for the system, applications and services are stored in the */etc* file system. They contain the settings for the system and the services. Examples are */etc/ssh* directory for the Secure Shell (SSH) service configuration files, */etc/shadow* file for the configured users on the system, */etc/group* file for the groups on the system. The content of these files in the /etc directory are an important point for investigation because they can be altered by attackers. Unexplained alterations to these files are an indication of compromise. Examples are unexpected user entries and blank root password in */etc/shadow* as well as changes in */etc/ssh* directory files making it more permissive for remote access (Nelson et al., 2014).

2.4 Malware Analysis

The study of malware specimen code and behaviour in an isolated environment is an important undertaking to get a complete understanding of the extent of damage it may have caused during a breach. This study also assists in prevention of repeat attacks especially from variants and incarnates of the one under analysis. Working on the assumption that a lot of existing malware are a result of polymorphic and metamorphic variants of older ones, analysis can provide protection vectors beyond the signature based detection strategies (Bayer, Kirda, & Kruegel, 2010).

Malware analysis is a key function of security research organisations, malware prevention software vendors and information security incident response firms. Anti-virus signature updates, software patches and anomaly heuristics engines are products of malware analysis (Sikorski & Honig, 2012).

2.4.1 Static Analysis

Static analysis is a form of code analysis that requires an in depth knowledge of CPU architecture and instruction as well as programming (Sikorski & Honig, 2012). This form of analysis is a walkthrough of malware source code if available although some disassembly, debugging and sometimes decryption might be necessary. Static analysis can be applied to different forms of programs, source code and program binary. A common method of static analysis involves function parameter analysis speculating on possible values of function arguments values and types to predict possible effect and consequences. Another technique for analysing binary representation of code involves identifying library functions and the point they are called in the code. From these function call graphs, the intent of the code can be discovered (Egele, Scholte, Kirda, & Kruegel, 2012).

Static analysis has the advantage of being impervious to conditional code execution and file level tweaks that can thwart or influence dynamic analysis (Bayer et al., 2010). This is because all modules and functions of the code can be stepped through and explored during the analysis. Apart from the manual intensity of static analysis, steep learning curve and specialised knowledge required, the other limitation of static analysis is that it comes unstuck in the presence of code obfuscation, runtime packing and encryption (Tian, Islam, Batten, & Versteeg, 2010). It struggles with packers that self-

modify code as well as code that depend on dynamic values like current date and time (Egele et al., 2012).

2.4.2 Dynamic Analysis

Dynamic malware analysis involves code execution in a secure and isolated environment (Egele et al., 2012). This execution can be a virtual machine or isolated system. The analysis involves an examination of the behaviour of the malware. It has the advantage of being unaffected by runtime packing and code obfuscation. Dynamic analysis lends itself to automation and scripting for large scale analysis (Oktavianto & Muhardianto, 2013). Dynamic analysis monitors the system under investigation for system calls, filesystem changes, dynamic link libraries and modules, logs and registry file changes while malware execution is ongoing. The result of changes observed give an indication of the workings of the malware under observation (Willems, Holz, & Freiling, 2007).

Its shortcomings include possible failure to explore the full gamut of the malware if there are conditional hooks in the code preventing it from executing a specific procedure because of an unmet condition (Bayer et al., 2010). There is also a limitation of time. Some malware might not run all the time or within a specific time interval. The foregoing make it possible for dynamic analysis to miss characteristics of the malware sample because a single run might turn out to be inadequate to evaluate the full impact (Provataki & Katos, 2013a). Other challenges to dynamic analysis are anti-analysis features like sandbox detection and anti-dumping (Zoltan, 2016) and (De Andrade, De Mello, & Duarte, 2013).

A technique for dynamic analysis is use of function hook to record the activities of functions called by malware at a high level. These functions are usually a part of APIs and system calls. This method will not work for a kernel mode malware. Where there is source code access, hooking can be done by specifying the hooking function where the monitored calls are invoked. For binary access, debuggers can be used to monitor the program execution by adding breakpoints to the call instructions or monitored functions giving debugger access to the memory areas and the state of CPU registers for the running process. Another function hooking method, for dynamic analysis, involves manipulating the binary execution in such a way that calls to the monitored library functions are passed to the hooking functionality first by manipulating the virtual addresses of call instructions in memory so instead of executing the instructions in at those addresses, the hooking functions is invoked instead, giving visibility to function parameters and arguments (Egele et al., 2012).

2.5 Related Work

This section is an overview of existing studies related to evaluation of open source sandboxes for analysing Linux malware. These works can be divided into works that evaluate existing malware analysis solutions and those that do an analysis of Linux malware samples. The former explores the position of sandboxes in malware analysis process, surveys of malware sandboxes and other tools in the analysis process. The latter is a review of analysis of existing research activities on analysis of Linux malware samples.

2.5.1 Survey of malware analysis solutions

The subsections below are a review of existing literature that have undertaken a survey or an evaluation of malware analysis solutions. This discussion is divided into the components of malware

analysis systems, the explosion of malware variants - metamorphic and polymorphic malware, the effects of anti-analysis techniques and the implementation of sandbox solutions with respect to indicators of compromise.

2.5.1.1 Components of analysis systems

(Wagner et al., 2015) listed the components of malware analysis systems as data providers and analysis environments. The data providers are standalone tools or packages used for static and/or dynamic analysis of malware samples. These are tools such as code debuggers, like GDB, IDA, Radare2 used for static analysis and Volatility, Rekall for memory (dynamic) analysis. Packages are self-contained analysis environments like Cuckoo and ThreatAnalyzer. Dynamic analysis is reliant on the analysis environment which can be bare metal, virtual machines or emulated environments. (Wagner et al., 2015) reviewed an on-site 2010 installation of Anubis, FireEye MAS 6.40, 2013 version of Joe Sandbox, Process Monitor 3.1, API Monitor v2 r-13. The online public front ends of Malwr (Built on Cuckoo), ThreatAnalyzer and Anubis were also evaluated. The data samples for evaluation were the Windows portable executable and dynamic link library files, Uniform Resource Locators (URL), Portable Data Format (PDF) files. The data providers were tested for their ability to accept single and batch file submissions, analysis environment support – bare metal, virtual machines or emulated hypervisors. The analysis operations capabilities (file system, Internet and simulated networks services, system calls) of the tools were also examined as well as the reporting options available.

The capabilities of malware analysis systems can be extended with the addition of machine learning libraries. These serve to train systems for malware detection and classification. The output of data providers were used with the machine learning algorithms to build these systems (Shah & Singh, 2015). (Shah & Singh, 2015) and (Boukhtouta, Mokhov, Lakhdari, Debbabi, & Paquet, 2016) in their experiments used the output of dynamic analysis of malware samples and benign files to train systems for automated malware detection and classification. In (Shah & Singh, 2015), the extraction of prominent API calls from the execution of benign and malicious files were used as input to linear Support Vector Machines (SVM) library. (Boukhtouta et al., 2016), using the ThreatTrack online sandbox for deep packet analysis and examination of flow packet headers in malicious network traffic, trained a system using J48 machine learning algorithms. Benign traffic was sought from the Internet Service Provider edge and customer traffic.

(Egele et al., 2012) undertook a survey of dynamic malware analysis tools and sought to evaluate them using mode of analysis – user mode, kernel mode, full system simulation or emulation, virtual machine monitoring etc. Process level, API calls, network, file system operations support was also examined. Some of the systems evaluated are Anubis, Joebox, CWSandbox, Norman Sandbox and Ether.

2.5.1.2 Malware variants and automated analysis

The need for automated analysis of malware samples was also raised in (Shah & Singh, 2015). This was in response to the impracticality of the manual analysis with respect to the rate of malware samples being discovered and the possibilities afforded by the alteration of existing samples through polymorphic and metamorphic means. These variants are undetected by signatures written for the original samples. Polymorphic variations are created by inclusion of mutation engine in malware. This engine serves acts on the rest of the payload to produce the original sample at run time. The engine logic can sometimes involve encryption/decryption. In metamorphic alterations, the executable code is adjusted by adding dud functions or re-arranging existing functions. Both methods serve to render

pattern matching discovery ineffective (Shah & Singh, 2015). Metamorphic variants are more difficult to discover because while in polymorphism, the executed code is exactly same in content (when tested with cryptographic hash functions) as the original sample after the action of the mutation engine, metamorphism introduces changes to code that do not affect the functionality but the appearance (You & Yim, 2010). The foregoing is the motivation behind research works aimed at automation of the analysis process, detection of similarities between malware samples and introduction of machine learning libraries into the analysis process.

(Sarantinos, Benzaïd, Arabiat, & Al-Nemrat, 2016) examined the efficiency of fuzzy hashing techniques to detect similarities between different malware samples. This technique involves a process known as Context Triggered Piecewise Hashing (CTPH) which is a combination of traditional cryptographic hashes, rolling hashes and piecewise hashes. SSDEEP proposed by (Kornblum, 2006) was found to fast and effective with respect to running time in the absence of unspecialised hardware. SDHASH was found to be the most effective in malware detection. Other tools evaluated were mvHASH, MRSH v2. (Azab, Layton, Alazab, & Oliver, 2014) demonstrated the use of the Trend Locality Sensitive Hashing (TLSH) algorithm to group binaries of similar variants together. It was found to be almost as effective as SDHASH. NILSIMSA and SSDEEP were the other hashing techniques compared. The TLSH algorithm uses the K-Nearest Neighbours (K-NN) algorithm, a simple supervised machine learning classification algorithm for grouping objects based on similar training instances in the feature set.

(Choudhary & Vidyarthi, 2015) proposed a system for detection of metamorphic malware by using the output of dynamic analysis to a text mining function implemented using Support Vector Machine from WEKA machine learning suite. This was used to build a classifier using the process states of malware samples and benign files as training data. The analysis was done using the Process Monitor tool from Microsoft SysInternals suite.

(Nataraj, Yegneswaran, Porras, & Zhang, 2011) and (Han, Lim, Kang, & Im, 2015) proposed workflows that incorporated repeatable and automated processes by converting malware binaries into images and graphs with the resulting classification based on the output image texture. Benign files and malware samples were put through this process. The malware samples used for training already had classification from VirusTotal and Kaspersky Antivirus engine. The test of effectiveness was done with dataset from VX-heavens. This method is based upon work by (Nataraj, Karthikeyan, Jacob, & Manjunath, 2011) aimed at malware classification at scale using image texture. The research noticed that for grey-scale images, samples from the same malware family exhibited similar patterns and texture.

(Sebastián, Rivera, Kotzias, & Caballero, 2016) presented AVCLASS as a solution to the challenge of associating malware samples to a family. This tool removes the reliance on generic names when labelling a malware sample or identifying its family. It leverages on the combination of the malware sample size in the VirusTotal repository as well as the participating malware detection engine for which the repository aggregates output. The label assignment is based on the labels adopted by the detection engines. The solution has been evaluated on ten datasets of 8.9 million malware samples.

2.5.1.3 Anti Analysis

(Ferrie, 2016) acknowledged the importance of emulators to dynamic malware analysis classifying emulators into hardware, hardware assisted and software emulators. Malware analysis evasion techniques and possible mitigation activities were catalogued. The environments used in the

experiment were VirtualPC, VMware, Parallels, Bochs, Hydra and QEMU. Software emulators such as QEMU were identified as most transparent with the ability to cause uncertainty within the logic of the evasion techniques about the presence or otherwise of emulation. The use of software emulators can when tested by anti-analysis logic give similar output as routine CPU errors making detection logic unreliable. The observation of the growing use of cloud infrastructure with more systems being deployed on emulated virtual environments also reduce the impact of malware evasion techniques as malware will have fewer platform options and smaller execution footprint for operation if anti-virtualisation procedures are included.

(Kirat, Vigna, & Kruegel, 2014) proposed BareCloud which unlike other automated malware analysis systems, runs on transparent bare-metal native operating system analysis environment. It incorporates different analysis platforms like Ether in bare metal form and Anubis in emulated QEMU environment and Cuckoo using Virtualbox for virtualisation based malware analysis. It does not rely on snapshots taken before and after as it does not performed monitoring within the guest analysis client making it unable to detect non-persistent changes. While it can detect anti-analysis behaviour, it is unable to force malware execution.

Using the Trojan upcliker.exe, (Mehra & Pandey, 2016) tested the effectiveness of sandboxing techniques in presence of malware anti-sandboxing in form of Human Computer Interaction (HCI). Anubis and Malwr were tested with a decidedly unnamed proprietary suite used as control. While Anubis did not discover the malware, Malwr could detect changes to the file system and identify the Trojan as malware. The commercial tool had a more comprehensive report.

(Yokoyama et al., 2016) in their reconnaissance on online sandboxes, sent malware samples to online sandboxes with the objective of obtaining information about their internal operations to prevent malware from executing in the presence of those unique conditions. Virtualisation was used to allow creation of snapshots for pre- and post-execution comparison. The research focussed its discussion on windows because of its popularity.

In (Provataki & Katos, 2013b), it was concluded that different executions of a malware sample yielded different results. This test was done comparing a local Cuckoo installation with online malware analysis sandboxes such as Anubis (Anubis, 2015), GFI Sandbox (GFI Software, 2017), Comodo and Cuckoo-based Malwr (Malwr, 2016) by dropping the Trojan '*Trojan-Dropper.Win32.Xpaj.a*'. Anubis and Cuckoo identified the malware while GFI and Comodo did not. Multiple runs of the malware on the local Cuckoo installation with three Windows operating systems as the sandboxes revealed different file changes with each run.

(Noor, Abbas, & Shahid, 2018) proposed the Analysis Evasion Malware Sandbox based (AEMS) based on Cuckoo analysis sandbox. This system is specifically targeted at malware written to evade analysis. This system modified Cuckoo with the inclusion of a dynamic link library on monitoring station to allow for the detection of anti-analysis techniques. When these techniques are discovered during analysis of a malware samples, the execution of the malware sample is forced.

2.5.1.4 Sandboxes and Indications of compromise

Sandboxes are isolated environments for the execution of malware samples for analysis and research. They form important components of malware analysis service and malware security appliances. Malware analysis services are a combination of sandboxes offered as a service portal for receiving

malware submissions. They can be public or private portals. These services can share samples and they use cryptographic hash functions to uniquely identify samples. Malware security appliances are used to protect end points by dynamic analysis of unknown samples with embedded sandboxes (Yokoyama et al., 2016). (Vasilescu, Gheorghe, & Tapus, 2014) demonstrated a distributed firewall solution integrated with Cuckoo (Cuckoo Foundation, 2015) for automated malware analysis of malicious Uniform Resource Locators (URLs). Three virtual machines were spawn for the automated testing – Ubuntu, Windows XP and Windows 7. It was found that the results of automated analysis was comparable to that derived from performing a manual analysis using a Windows XP virtual machine running Volatility (The Volatility Foundation, 2014) for memory analysis, DumpIt (Suiche, 2016) for memory acquisition and IDA (Hex-Rays, 2016) for code disassembly but considerably faster. Willems, Holz, & Freiling, (2007) demonstrated CWSandbox, an automated dynamic malware analysis platform for WIN32 family of malware using API hooking and DLL code injection to avoid detection. It saves the state of the sandbox system before malware execution and compares this state to the post execution state. A plugin Cuckoo profiler was added to the standalone Cuckoo installation to speed up the discovery of the changes in the guest operating systems. The design and implementation of *TTAnalyze* was described in (Bayer, Kruegel, & Kirda, 2006). This system used the QEMU emulator to monitor Windows system and API calls made by the malware sample under observation.

Another approach to automated malware analysis makes uses of CPU virtualisation extensions and purpose built virtual machine monitors running at a higher privilege level by booting with the host operating system. These monitors can intercept system calls on the guest operating systems used for malware analysis transparently to avoid detection by malware. This method was used in (Dinaburg, Royal, Sharif, & Lee, 2008), (Nguyen et al., 2009) and (Deng, Xu, Zhang, & Jiang, 2012). (Dinaburg et al., 2008) demonstrated Ether, a solution based on the Xen hypervisor that used the Intel VT CPU virtualisation extensions to monitor Windows API and system calls during malware behaviour analysis. (Nguyen et al., 2009) implemented a dedicated virtual machine monitor for malware analysis MAVMM using the AMD SVM as hardware virtualisation to monitor malware behaviour on Ubuntu 8 guest operating system. Using ninety-three (93) real world Windows viruses, (Deng et al., 2012) implemented *IntroLib* using the KVM hypervisor to intercept the interaction between malware code and operating system library code. Based on the same principle used in Ether and MAVMM, *IntroLib* was found to have lower overhead.

The use of a combination of static and dynamic analysis output as learning features for machine learning libraries was employed in (Islam, Tian, Batten, & Versteeg, 2013) and (Shijo & Salim, 2015). In the former, static examination was done using IDA pro to get total length of all functions in the malware executable as well as printable string information. These features were combined with dynamic analysis features from API logs using the API feature extraction tool from the virtual environment. An application was built to interact with the WEKA machine learning library using the features from the analysis to classify malware. The malware samples were obtained from CA Vet zoo. Using malware samples from VirusShare and dynamic analysis performed on Cuckoo sandbox running on Ubuntu 10 with Windows guests on VMware, the latter extracted API call logs and combined with static features in form of printable string information obtained using the *strings* utility, developed a classification system using SVM and random forest algorithms.

(Neugschwandtner, Comparetti, & Platzer, 2011) implemented SQUEEZE integrated with Anubis to observe Domain Name System (DNS) traffic during dynamic analysis of malware samples. The objective was to detect failover activities of Command and Control (C2) servers in botnet attackers in the face of takedown attempts. It explored the execution path malware explores when the primary

path is blocked. The logic for failover is run by ensuring a forced execution of malware domain generation algorithm.

With dataset from VirusShare, (Tirli, Pektas, Falcone, & Erdogan, 2013) presented Virmon as an analysis system that is effective for current Windows operating systems on 64-bit architectures. This is in response to the trend of testing malware with older Windows operating systems on 32-bit architecture. Virmon analyses malware samples by monitoring host based features like file system, registry, process interactions and network features such as DNS requests that the malware droppers might be making to reach the command and control centres. (Pektaş & Acarman, 2017) proposed a system for classification of malware families based on runtime behaviour. This system made use of Virmon and Cuckoo sandbox to extract run time behaviour features - API calls and registry changes. Malware sample labelling was done with VirusTotal with the samples divided into training and test set.

(Aslan & Samet, 2017) tested local and online sandboxes in conjunction with standalone static and dynamic analysis tools like PEiD, PEview, PE Explorer, MD5Deep, Process Explorer, Process Monitor. The local sandbox installations tested are Cuckoo, CW Sandbox, Norman Sandbox, Droidbox, while the online sandboxes tested are VirusTotal, ThreatExpert, Malwr. Jotti's Malware Scan. The malware samples were tested on Window 7 and Windows 10 Virtual machine instances. It was found that using a combination of these tools yielded better results than individual use of the tools. The best combination comprised of Cuckoo sandbox, IDA Pro, PEiD, PEview, Process Monitor, Wireshark, Malwr and VirusTotal.

(Tsyganok, Tumoyan, Babenko, & Anikeev, 2012) sought to classify polymorphic and metamorphic malware samples based on the features exhibited during dynamic analysis. Cuckoo framework with Windows virtual machine guests was used for classification of Windows portable executable files. Cuckoo was used to extract operating system and network interactions, file system changes and code injections.

2.5.2 Analysis of Linux Malware Samples

Machine learning methods were applied to detecting Linux malware in (Mehdi, Tanwani, & Farooq, 2009), (Shahzad & Farooq, 2012) and (Asmitha & Vinod, 2014). While (Mehdi et al., 2009) investigated the process memory image and used the task struct data structure as classification criteria, (Shahzad & Farooq, 2012) and (Asmitha & Vinod, 2014) investigated the ELF file structure. The former based its classification on 383 features of the ELF file headers and using 709 Linux malware samples from VX-heavens (VX Heaven, 2017) recorded 99% accuracy in malware detection without resorting to signatures, making it effective for zero-day malware detection. The latter used the systems calls invoked during the execution of an ELF file as classification criteria and this scheme recorded 97% accuracy using Linux malware samples from VX-heavens.

(Pa et al., 2015) proposed IoTPOT and IoTBOX targeted at malware analysis of Internet of Things (IoT) devices. The former is high interaction honeypot supporting multiple architectures and IoTBOX is sandbox supporting cross compilation to multiple platforms using QEMU for device emulation. IoTPOT as a honeypot emulates Telnet services of various IoT devices to allow detailed analysis of an on-going attack. IoTBOX, using QEMU emulator supports analysis of malware on eight (8) different CPU architectures, namely as MIPS, MIPSEL, PPC, SPARC, ARM, MIPS64, sh4 and X86.

(Damri & Vidyarthi, 2016) in their survey of techniques used for dynamic analysis of malware samples written for the Linux operating identified five (5) approaches. These approaches are based on the system call, process control block, ELF, kernel and hybrid investigation which is a combination of any of the other four methods. The investigated literature used *strace* tool to investigate the system calls, arguments and return values. The process block is the runtime state of the process in *task_struct* containing the user and group identity of the process owner, memory information, file system information, process signals, open files and network sockets. Features of the ELF header in the execution state were used in some of the surveyed literature to differentiate between malicious and benign programs. Files in the */proc* file system like the */proc/meminfo* and */proc/cpu* were used to extract information about the kernel state which some of the evaluated works used to detect the presence of malware.

(Asmitha & Vinod, 2014) proposed a system call based investigation technique for malicious activity detection. This involved a system call logger and extraction of useful function calls. This was used with malware from VX-Heavens. The malware samples were divided into training set alongside benign samples from the /bin directory and test set to ascertain the effectiveness of the algorithm.

With 10,548 Linux ELF malware samples sourced from VirusTotal, (Cozzi, Graziano, Fratantonio, & Balzarotti, 2018) identified challenges associated with Linux malware analysis such as variety of possible CPU architectures, deliberate obfuscation techniques such as file header manipulation and packing. Challenges with shared libraries and dynamic linking were also mentioned. The samples were subjected to a series of processing steps involving investigation of file metadata, static and behavioural analysis. The *file* utility was used to determine the file types, the *readelf* binary was used for file metadata analysis and AVCLASS (Malicia Lab, 2018) was used to label the samples by antivirus families. IDA Pro debugger (Hex-Rays, 2016) and Radare2 (Radare, 2017) were used for the static analysis step while dynamic analysis was carried out using SystemTap (Sourceware, 2018b) to investigate file system interactions, systems calls during execution of the samples in KVM (Openshift, 2018) and Qemu (QEMU, 2017) CPU emulators. Malware samples written for x86, x86-64 were executed with the KVM emulators while 32-bit MIPS, PowerPC and ARM architectures were executed using Qemu. The experiments revealed the differences within malware families in terms of obfuscation techniques and access privileges required to run the malware samples. The methods malware authors employ to ensure persistence of malicious code and the frequently requested system calls, libraries and commands were also determined.

Curated lists of automated malware analysis sandboxes and online services from (Shipp, 2018) and (Zeltser, 2017a) have only five tools for Linux malware analysis from a combined list of 42 tools. The five tools are REMnux which is used in SANS malware analysis course (SANS, 2017), Cuckoo sandbox Limon, Detux and Tencent HaboMalHunter. (Monnappa, 2015) presented Limon at the Blackhat Europe 2015 conference (UBM Tech, 2015). In the conference, the static and dynamic memory analysis of the *tsuna/httpd.txt (voip scanner - binario elf)* ELF malware sample was demonstrated. Tencent HaboMalHunter sandbox is the open source standalone version of the Tencent Habo analysis portal that has been integrated with VirusTotal (VirusTotal, 2017). It is a subproject of the Tencent Malware analysis platform (Tencent, 2018).

2.6 Research Goals

The previous section is a summary of the current literature and the contributions they have provided. The discussion of the indications of compromise and the use of sandboxes for malware analysis has

been dominated by research related with analysis of Windows portable executable malware samples. In the exception above, (Vasilescu et al., 2014) analysed malware on an Ubuntu guest to show that employing sandboxes for automatic analysis is faster and as effective as manual analysis. The existing research on Linux malware analysis have been predominantly focussed on the use of individual tools or data providers like *readelf*, *strings* utility and *strace*. The output of these tools was used as training data for machine learning algorithms and libraries.

The objective of this research is an investigation of the malware analysis sandboxes that are known to support Linux ELF binary malware analysis to determine their relative capabilities. This research is focussed on the issues raised in the previous section with respect to the evaluated sandboxes' effectiveness in detecting compromise, resistance to anti-analysis as well as the level of support for automation and reporting. The relative effectiveness is evaluated, considering the static and dynamic analysis capabilities of the tools. Static analysis is used to determine if samples match signatures of known malicious files and if anti-analysis tools such as packers and morphing engines are used. Dynamic analysis determines the ability of a sandbox to spot indications of compromise after memory analysis, and investigation of network traffic and operating system interactions such as system calls and file system operations. The reporting and automation support is also investigated.

From the previous subsection, five sandbox environments were identified and this research focuses its comparison on the five packages. They are Cuckoo, Limon, REMnux, Detux and HaboMalHunter. The analysis is done with the latest Linux kernel version – 4.4. In the next chapter, the approach to this comparison will be undertaken with a discussion of the research design decisions and the testing methodology.

3 Research Design

3.1 Introduction

This chapter details the research methodology and the reasons for the design decisions. Malware analysis requires acquisition of data samples for testing and a process for testing (Wade, 2011). The next section is a discussion of the approaches other researchers have employed in the acquisition and testing of malware samples. It follows from the concluding sections of the previous chapter that reviewed the related literature on the theme of malware analysis tools. The third section highlights the approaches adopted for acquisition of malware samples in this research. The testing process is described in the fourth section. A restatement of the research objectives concludes this chapter.

3.2 Review of malware analysis methodology

Testing malware and evaluating malware analysis sandboxes requires access to malware samples and the implementation of a testing environment. This section reviews the approach similar research efforts have taken to source malware samples as well as the methods used for testing the malware samples.

3.2.1 Sourcing malware samples

Two approaches have been used in the acquisition of sample data for malware analysis. The first approach involves the use of honeypots and entrapment systems. These are systems that are designed to attract potential attackers to get an understanding of contemporary threats at the time of deployment (Guarnizo et al., 2017). They can be classified based upon the level of interaction permitted (Mokube & Adams, 2007) or by deployment purposes (Mairh, Barik, Verma, & Jena, 2011). With respect to level of interaction, they can be further classified as high or low interaction honeypots. The purpose of deployment can be research related or as a permanent fixture on a production network to serve as an early warning device for detection of vulnerabilities and exploits. (Rieck, Holz, Willems, Düssel, & Laskov, 2008) and (Vasilescu et al., 2014) employed spam traps and honeypots to acquire malicious emails attachments and URLs in their investigation of patterns for classification of malware and comparison of manual and automated malware analysis respectively. In their investigation of malware samples that they considered as being under-investigated (64-bit Windows portable executables and Dot Net/Mono files), (Botacin et al., 2017) extracted sample Dot Net files from email attachments.

The other method for sourcing malware samples requires the use of public and private malware repositories. These are portals that accept malware samples from members such as researchers, anti-malware vendors and partners. They classify these samples and generate cryptographic and fuzzy hash functions of these samples to detect if they are the same or similar samples to previously uploaded ones. They provide search and download functionality of varying access levels to different categories of users. Some require membership or registration (free or fee-based) (Zeltser, 2017b) (Shipp, 2018). In the research by (Mehdi et al., 2009) and (K. Asmitha & P. Vinod, 2014), the training data for the malware processes that formed input to the machine learning libraries were sought from VX Heaven (VX Heaven, 2017). Malware samples from VX Heaven and Offensive Computing (Offensive Security, 2017) were used by (Shahzad & Farooq, 2012), (Deng et al., 2012) and (Shahzad, Shahzad, & Farooq,

2013) in their investigation of Linux ELF binary headers, operating system calls and kernel data structures respectively .

More than 38,000 malware samples from Virus Share (Virus Share, 2017) repository were used as training data in the demonstration of an open source automated malware analysis tool by (Rubio Ayala, 2017). In building a stable, high processing malware analysis platform, (Miller et al., 2017) used tens of thousands of malware from VirusShare with various parameters and virtualisation systems. These tests were done to compare stability of platform when different parameters and virtualisation platforms were used. The specifics of the malware samples were not of significance in these tests. The volume of samples and the effect on the tested platform choices were the most important considerations. (Botacin et al., 2017) obtained the 64-bit Windows PE samples from VirusShare in their investigation of non-mainstream malware samples. Samples from VirusShare were also used by (Gandotra, Bansal, & Sofat, 2016) in their experiment aimed at training a system to detect zero-day malware attacks.

3.2.2 Analysis methods

The analysis methods used in the existing body of work are a combination of static, dynamic and hybrid analysis techniques. (Mehdi et al., 2009) and (Shahzad et al., 2013) employed dynamic analysis by observing malware process execution using *strace (Strace, n.d.)*. (Shahzad & Farooq, 2012) made a static analysis of ELF headers of malware samples. The hybrid approach combining static and dynamic approaches was employed in some other research activities. The approach makes use of sandboxes and virtual machines to execute malware samples, studying the memory, processes and network interactions, while also examining the binary code (header and routines) for signs of packing and obfuscation. The static analysis component also submits cryptographic hash checksum output to online repositories like VirusTotal (Vasilescu et al., 2014). VirusTotal accepts malware samples from individual contributors, researchers, anti-malware vendors and other sources. It has a database of malware samples, cryptographic hash function outputs and a reporting system indicating the anti-malware engines that have successfully identified a file sample as malicious as well as the engines that have failed to detect the samples as malicious (Google, 2017).

(Vasilescu et al., 2014) tested malware samples using Cuckoo sandbox integrated with a distributed firewall application implemented on an Ubuntu system for its automated tests and the combination of VirusTotal submissions and queries, Wireshark, IDA debugger and DumpIt for manual analysis. Other works that utilised the Cuckoo sandbox are (Rubio Ayala, 2017), (Miller et al., 2017) and (Gandotra et al., 2016). (Rubio Ayala, 2017) employed Cuckoo sandbox in combination with Weka machine learning libraries to demonstrate an open source malware analysis system. Cuckoo sandbox and the Weka machine learning libraries were also used by (Gandotra et al., 2016) in implementation of a system to detect zero-day attacks. In their demonstration of a stable high processing malware analysis system, (Miller et al., 2017) tested Cuckoo sandbox with KVM, VMware and Virtualbox virtualisation platforms using different testing options. CWSandbox was used in (Deng et al., 2012) and (Rieck et al., 2008). The former proposed a high-performance malware detection system immune to kernel hooking by malware – *Introlib* which was compared with CWSandbox and Anubis while the latter used CWSandbox to implement a system for the study and classification of malware samples.

Table 3.1 is a summary of the research design decisions of related works.

Table 3. 1 Summary of design decisions of related research efforts

Author(s)	Aim/Outcome	Malware Sample/Source	Analysis Tool(s)/Method
(Rieck et al., 2008)	Detection, Investigation and classification of malware	Spam trap and honeypot	CWSandbox
(Vasilescu et al., 2014)	Compare manual and automated malware analysis	Malicious URLs and email attachments	Cuckoo and integrated Firewall compared with manual analysis using combination of Wireshark, VirusTotal, IDA Debugger, Dumpit
(Hirono, Yamaguchi, Shimada, & Takakura, 2014)	Trace effect of malware from inside a network	Poison Ivy RAT	Python network services libraries for Internet services emulation, Squid proxy for transparent proxy, Clam AV and Snort IDS for threat analysis
(K. Asmitha & P. Vinod, 2014)	The goal is to detect metamorphic and polymorphic viruses	226 malware samples from vxheavens and 442 benign files from /bin, /usr/bin, /sbin	Strace to monitor binary execution
(Mehdi et al., 2009)	Detect parameters in process task struct that differ greatly between benign and malicious processes	VX heaven	Investigation of task_struct data structure in kernel
(Shahzad & Farooq, 2012)	Creation of non-signature based malware detection scheme	VX heaven and Offensive Computing	ELF header binary investigation
(Deng et al., 2012)	Creation of high performance dynamic analysis virtualisation tool immune to API hooking and emulation detection	Sample from Offensive computing	KVM based hardware virtualisation platform - Introlib which intercepts library calls on Windows and Ubuntu Linux 11.04 guest platforms. compared with Anubis and CWSandbox
(Shahzad et al., 2013)	Creation of kernel module for differentiation of benign and malicious processes	Training data included 114 malware samples from VX heaven and offensive computing and 105 benign files	Investigation of kernel structure of processes
(Rubio Ayala, 2017)	Creation of open source software for malware analysis	54 software programs from CNET Download site and 549 freeware.com formed the benign group while 97 Windows System PE malware and 38152 CrptoRansom malware were sourced from VirusShare	Cuckoo, Virtualbox and Weka data mining and machine learning library.
(Miller et al., 2017)	Build stable high processing dynamic analysis platform	Various tests using tens of thousands of samples from VirusShare. The number of sample in this case is more important than the specifics of the samples as the stability of platform with different associated software and settings were the goal of the exercise.	Cuckoo with various settings and virtualisation platforms (Vmware, Virtualbox and Qemu)
(Botacin et al., 2017)	Analysis Dot Net/Mono and 64-bit Windows Portable executables	426 samples (from suspicious e-mail attachments collected between 2012 and 2015), 64-bit Windows malware binaries from VirusShare	Qemu/KVM with monitoring driver to record callbacks, registry calls
(Gandotra et al., 2016)	Detecting zero-day attacks using machine learning trained system	Malware samples from VirusShare confirmed with AVG antivirus and benign files from Windows system directory	Modified Cuckoo with Weka data mining and machine learning libraries

The summary in Table 3.1 shows the approach other researchers have used in sourcing for malware samples and the testing environments for malware analysis. These research efforts with the exception of (Botacin et al., 2017) used either honeypots or malware portals to source malware samples. (Botacin et al., 2017) used both methods, however the analysis was not related to Linux ELF binaries. CWSandbox, Anubis and Cuckoo were the only malware analysis sandboxes used in the tests above. Anubis was an online portal for malware analysis, however it does not exists any more (Anubis, 2015). CWSandbox does not exist as a free use service any longer and it does not support the analysis of Linux ELF binaries (Ouchn, 2011). The reviewed literature in Table 3.1 generally employed malware repositories to determine and establish the baseline for malware samples which is used for comparison with the experimental results derived from the tests. This approach is adopted in this research in addition to the sourcing malware samples from a honeypot.

3.3 Data Acquisition

The use of malware repositories and a honeypot entrapment scheme was adopted for this research. The malware repositories offer labelling, identification and classification services for the malware under investigation to allow for consistent testing of the sandboxes. The use of the honeypot is to supplement the samples from repository with live malware samples and network packet captures, creating the possibility of adding unknown malware samples to the testing pool. This section discusses the setup and components of the honeypot as well as the selection and testing of the malware samples from the online repositories selected. The reasons behind the design decisions are also discussed.

3.3.1 Honeypot

The use of the honeypot creates the possibility of adding unknown malware samples to the pool. This section focuses on the setup, component description and decisions taken in the implementation of the honeypot. A high interaction honeypot was set up to allow the whole system to be infected with the aim of capturing and understanding the malware sample(s) and reusing them to test the sandboxes. Figure 3.1 shows the topology of the honeypot setup.

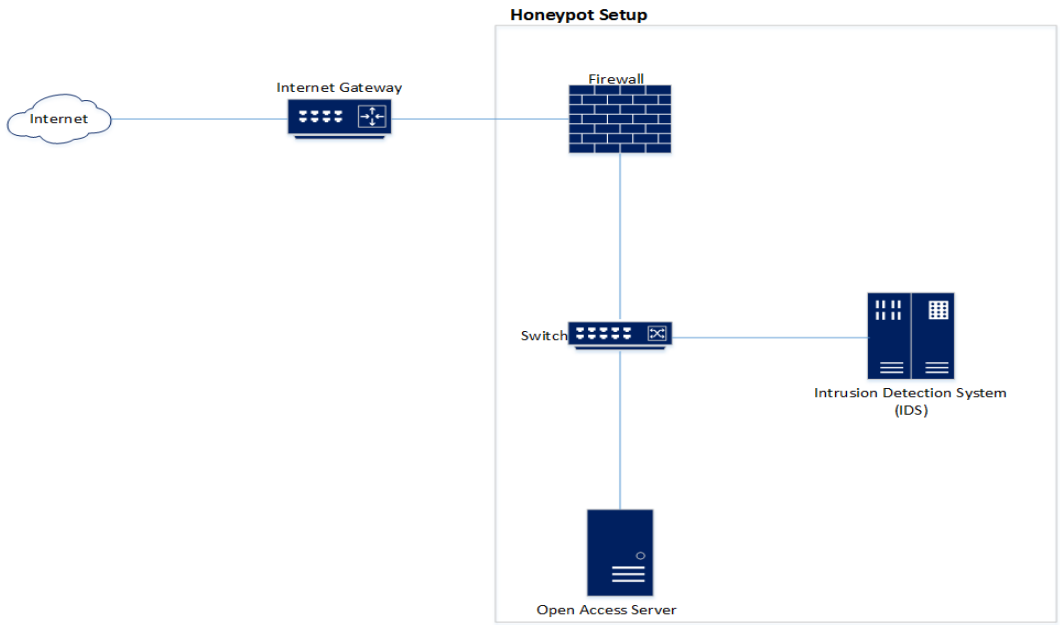


Figure 3. 1 Honeypot Topology

3.3.1.1 Open access system

The open access system is the honeypot system that is deployed to be infected and compromised. This system runs Ubuntu 16.04 operating system. This is the current Long-Term Support (LTS) distribution and it has maintenance support until 2021 (Canonical Ltd, 2017). The Ubuntu distribution was selected in particular because according to (DistroWatch, 2017), since its inception in 2004, it has grown to be one of the most popular Linux distributions. After installation, the system was updated to run the most current security and application updates and LTS supported kernel version 4.10. A web application was deployed on it with a MySQL (Oracle, 2017) database server backend. The web application was created using the Flask framework (Ronancher, 2017) and it was deployed on an Apache web server (The Apache Software Foundation, 2017).

The web application accepts input from a web form that requires the first name and surname of an actor or actress and returns in JavaScript Object Notation (JSON) format, the list of titles the actor or actress has appeared in and the name of the character portrayed. MySQL, being a networked database service also offers potential attackers an additional footprint to attack the system (Singhal & Ou, 2017). Flask was chosen because it offers rapid deployment and ease of web application prototyping (Grinberg, 2014). The choice of JSON output is also to give the impression that while the system is standalone, it could also be a part of a larger system that can use it for machine to machine communication in the form of an API. The popularity of the Apache webserver necessitated its use as the application web server as it is still the most deployed webserver (Netscraft Ltd, 2017) and it offers an additional attack surface for potential attackers (Durumeric et al., 2014).

The Advanced Intrusion Detection Environment (AIDE) (Sourceforge, 2016) application was deployed to detect file manipulation and changes to the file system, configuration files and system binaries. AIDE works by taking a snapshot of the attributes of files marked for monitoring. AIDE can subsequently detect manipulation of monitored files if changes are made either to the files with respect to content or access control rights and ownership. The snapshot of the system binaries and configuration files were taken. MD5 and SHA256 cryptographic hash function outputs were taken of the snapshot before it was transferred to the trusted upstream system. The hash values were confirmed at the upstream server to match the hashing function output obtained on the open access system prior to the transfer. The AIDE application was uninstalled and the configuration and library files deleted so that potential attackers are oblivious to the intrusion detection plans.

The Secure Shell (SSH) server configuration file - */etc/ssh/sshd_config* was edited to permit root login and the root password of 'love' was set to make the system easily susceptible to a dictionary attack. allowing easy access to the system from a brute force attacker. The username of 'root' has been found to be one of the most tried usernames for brute force attacks for remote login (Sochor & Zuzcak, 2015).

3.3.1.2 Firewall

A Juniper (Juniper Networks, 2017a) SRX 300 security appliance was deployed as the firewall. It was running Junos OS version 15.1X49 (Juniper Networks, 2017c) which is the tested version for the published specifications of five thousand (5,000) sustained new sessions per second, a maximum of sixty-four thousand (64,000) and one thousand (1,000) concurrent sessions and firewall rules respectively. It has also been benchmarked to have a firewall traffic throughput of 1Gbps (Juniper Networks, 2017d).

The SRX 300 has the property of stateful firewalls whereby the permitted connections are tracked so related transactions of same sessions are permitted without recourse to the CPU cycles of matching packets against firewall rules. For example, if a firewall rule permits traffic of a protocol type from a specific internal host to a destination external host, return traffic from the destination is also implicitly permitted without a need for a firewall rule match. The traffic headers are checked to confirm that they are part of the same session that had been tracked when the initial firewall rule was invoked (Gouda & Liu, 2005).

The firewall was partitioned into two security zones. The internal zone which the honeypot server was placed was named zone *in*, while the external zone (the expected source of attack traffic) was named zone *out*. The firewall rules deployed on the firewall effectively permitted access to the server (zone in) from the wider Internet, while denying traffic originated from the server to all external destinations (zone out) except Google public Domain Name Servers (DNS) 8.8.8.8 and 8.8.4.4. Allowing domain name resolution was done to increase the level of interaction available to prospective attackers. The deny actions were to silently discard packets without sending an Internet Control Message Protocol (ICMP) destination unreachable packet. Sending this packet would have made the presence of traffic filtering visible to an attacker (Rosen, 2014). Table 3.2 is a summary of the stateful firewall rules configured.

Table 3. 2 Firewall security policies

Source Zone	Source host/network	Destination Zone	Destination host/Network	Applications	Action
out	Any hosts	In	202.124.118.49	All applications	Permit and log
in	202.124.118.49	Out	8.8.8.8 and 8.8.4.4	UDP 53 DNS	Permit and log
in	202.124.118.49	Out	Any hosts	All applications	Deny and log

3.3.1.3 Switch

The connecting switch in this setup is a Juniper EX2200-C switch running Junos OS version 12.3. It has twelve (12) gigabit ethernet ports and supports full line rate switching (Juniper Networks, 2017b). It was configured for port mirroring. This enabled the duplication of all the packets entering and leaving the port assigned to the honeypot to a Virtual LAN (VLAN). The only member of this traffic monitor VLAN is the port the IDS system is connected to. Table 3.3 shows the port to host and VLAN mapping configuration of the switch.

Table 3. 3 Switch interface and VLAN configuration

Device	IP address	Port Number	VLAN ID
Open access system	202.124.118.49/31	ge-0/0/0	75
Firewall	202.124.118.48/31	ge-0/0/11	75
IDS	N/A	ge-0/0/1	999

3.3.1.4 Intrusion Detection System

The Intrusion Detection System (IDS) was deployed in passive mode to capture network traffic packets. It was implemented using an Ubuntu 16.04 LTS system with Snort IDS and TCPDUMP installed. Its network interface is connected to the switch to accept all communications coming from and going to the honeypot system. This is stored in PCAP format. Snort and TCPDUMP commands were used to place the interface in promiscuous mode to capture the packets.

3.3.1.5 Trusted upstream server

A Virtual Private Server (VPS) was deployed as the trusted upstream server. This is a Debian Linux server that is used to transfer to and copy files from the honeypot system. This server does not use password authentication and does not allow root login. SSH keys are required for client access to the server. The public key is stored on the server while the private key is on a remote client used to connect to the server. Communication is always initiated from the server to the open access server and not vice versa to avoid the need to have a copy of the private key of the secure key exchange on the honeypot system. This server was used to copy the AIDE database files from the open access system. It was also used to copy the *linpmem* binary (Cohen, 2016) for memory acquisition to the honeypot system.

3.3.2 Public Repositories

A number of repositories have been used in the research community as sources of malware as discussed in section 3.2.1. However, the need to use current malware samples as well as the specific constraints to analyse executable Linux malware file samples made some of the public repositories employed in previous research activities unsuitable. The latest bulk compilation on VX heavens was compiled in 2010. The quest for recent malware submissions (post 2014) stems from research that the properties of the samples uploaded prior to 2014 that might be of interest during testing will exist in some of the more recent samples since most new malware samples are polymorphic or metamorphic variants of old samples (Alam, Horspool, Traore, & Sogukpinar, 2015; Bist, 2014; Sharma & Sahay, 2014). The focus on recent samples allows the analysis sandboxes to be tested against the current field and trends in malware as well as variants of old samples.

Another repository that featured prominently in the related works discussion was Offensive Computing (later transitioned to Open Malware). This repository is no longer available. VirusShare repository was the only repository left from those used in other research activities. Some of other repositories based on the list from (Zeltser, 2017c) were explored. VirusTotal and VirusShare were found to meet the requirements of the research for recent (between 2014 – 2017) ELF binaries for the x86 and x86-64 CPU architectures. A total of two hundred and ninety-three (293) samples were acquired from the two portals.

3.3.2.1 VirusTotal

VirusTotal is a portal for malware submission and scanning. It generates a report for the submitted malware based on the aggregation of results from participating vendor anti-malware engines. Its samples come from the public as well as partnerships with antivirus companies. Reports can be generated by submission of suspected malware sample files directly or by using as query parameter, the cryptographic hash function output of a suspected malware sample. Its database has other

metadata such as the MD5, SHA1, SHA256 cryptographic hashes and the SSDEEP fuzzy hash of the samples. Other information about the sample include date of first submission, scan results of each of the engines used as well as the names the sample is known by the engines that have flagged it as malicious (Google, 2017).

The date of first submission of the sample was used as the indication of currency in the selection of malware. Only samples first submitted between 2014 and 2017 were considered for analysis. That was the primary use of VirusTotal in this research. It was used to get an indication of the age of a malware sample. The malware repository of VirusTotal could not be utilised because it required a paid subscription service. The paid subscription service also grants access to a private API key for making more detailed queries. A student research account was granted that gave access to forty-one (41) ELF malware samples out of which fourteen (14) were built for the x86 and x86-64 processor architectures. Those 14 samples were included in the analysis pool.

3.3.2.2 VirusShare

VirusShare required an account registration request and once approved, all the malware samples were available for download. There are special collections organised as torrent files but the ELF file collection was compiled in 2014. Most of the samples in it were first submitted to VirusTotal prior to 2014. The other bulk samples on the site were organised chronologically in compressed ZIP archive files. They are archives of various types of malware samples for different platforms. Each archive had sixty-five thousand, five hundred and thirty-six (65,536) samples.

The approach taken was to download from the latest bundle, extract the archive and run a Python script using the Unix magic file signature library to iterate through all the files to filter out the ELF files making a table of file name and magic file signature description. The *grep* utility was used to filter out the files built for the required processor architectures - the x86 and x86-64 processor architectures. This process was repeated for each bundle in succession. Two hundred and seventy-nine (279) samples were extracted for inclusion into the analysis pool at the end of this process.

3.4 Analysis methodology

This section discusses the analysis method used to evaluate the effectiveness and characteristics of the malware sandboxes being evaluated against Linux malware samples. There are two streams of analysis corresponding to the sources of the malware samples used. The malware sandboxes were evaluated against the malware samples derived from the VirusTotal and VirusShare repositories. The second stream involved investigating for indications of compromise on the honeypot system and extracting malware samples and traffic captures for further testing on the malware analysis sandboxes. Figure 3.2 illustrates these steps in the analysis process.

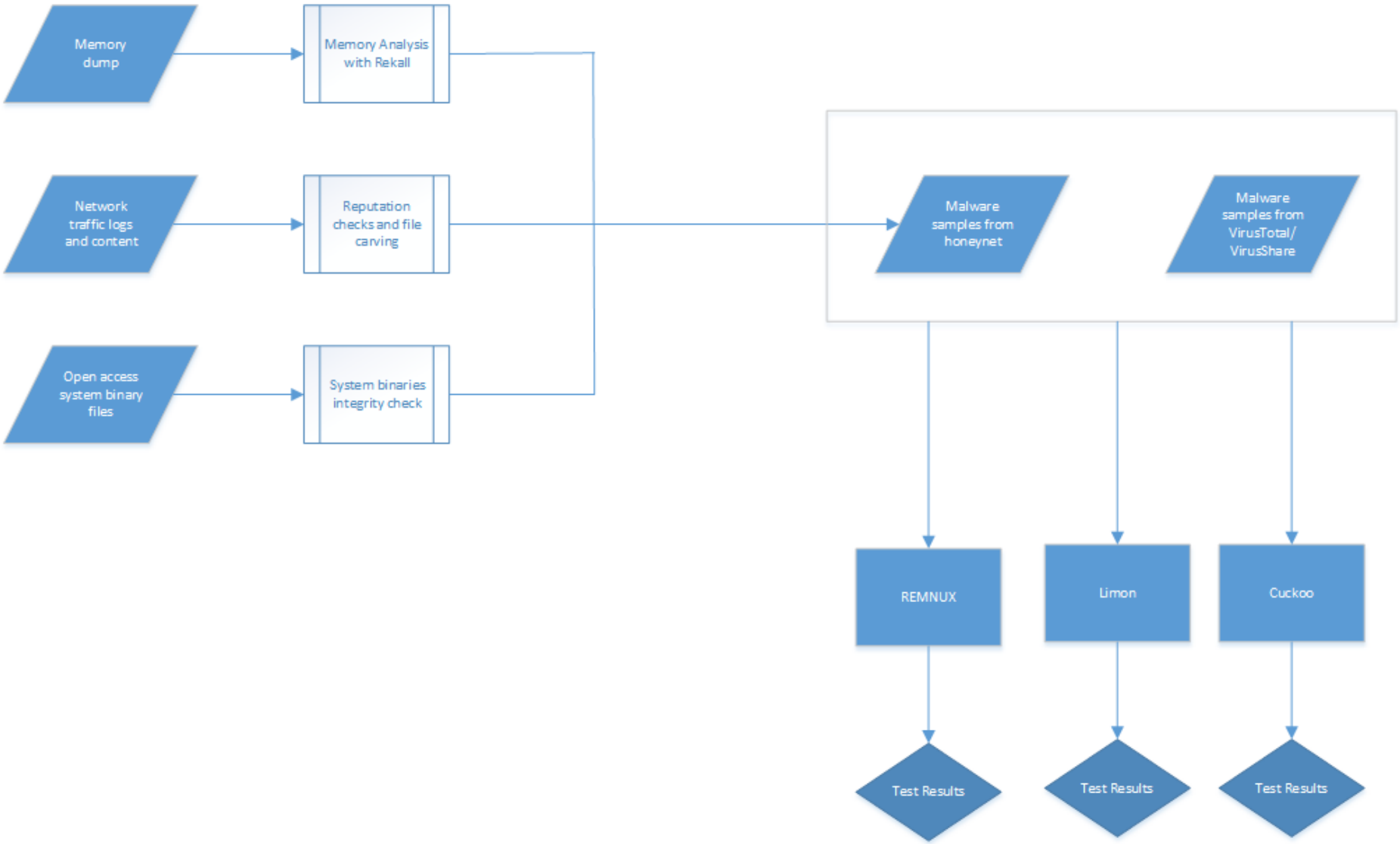


Figure 3. 2 Stages of Analysis

The sandboxes were tested for their ability to detect malware obfuscation techniques and the presence of malicious activities in the forensic artefacts. The automated analysis features of the tools and the supported reporting features were also evaluated. The specific anti-analysis technique that was tested involved capability of the sandbox to detect the presence of packers and metamorphic variations when analysing the malware samples. The effectiveness of the sandboxes for dynamic malware analysis was investigated by testing the ability of the sandboxes to detect malicious activities and indicators of compromise based on forensic artefacts like memory, network activities, operating system operations. Acceptance of batch jobs for the analysis of multiple malware samples was also tested to ascertain support for automated analysis.

3.4.1 Honeypot setup

The honeypot testbed presented an opportunity for additional malware samples to be added to the testing pool. An investigation of the file system integrity using AIDE was performed on the honeypot server. AIDE was reinstalled and the old database file was copied from the trusted server back to the honeypot server. A comparison between the old database (prior to allowing external access to the system) with the current generated output was done to determine the file system changes, additional and modified system executables. Network packet capture files from the firewall were examined and the reputation of each of the communicating IP addresses was checked on VirusTotal. The packet capture files were also examined for traffic attributes and content using Snort IDS.

The viable options for memory acquisition on 64-bit Linux systems were described in subsection 2.3.2.2. The options are the use of a loadable kernel module that is precompiled in the kernel of the target system or the use of the */proc/kcore* file if it is enabled. The */proc/kcore* file is a mapping of the physical memory of the system (Case & Richard, 2017). The use of a precompiled loadable kernel module was avoided as this can be detected by a potential attacker and serve as a sign that the system is a honeypot. A memory dump of the open access system was taken using the *linpmem* binary from the Rekall repository. This binary reads the /proc/kcore file which is enabled by default on Ubuntu and most Linux distributions (M. H. Ligh et al., 2014).

Rekall is a fork of the Volatility project and it has tools for memory acquisition and analysis while Volatility is solely a memory analysis tool. Rekall and Volatility have similar plugins for analysis of Linux memory images (Rekall Forensics, 2017). The decision to use to Rekall for the memory analysis was a consequence of the decision to use the *linpmem* binary from the Rekall toolkit for the memory acquisition. The default and recommended file format for the acquired memory image is the Advanced Forensic Format version 4 (AFF4) image file format (Cohen, 2016). This file format has support for storage of additional metadata about the system under investigation during analysis that can be utilised by the Rekall memory analysis plugins (Cohen, Garfinkel, & Schatz, 2009). The AFF4 format was used as the file format of the memory image. The Rekall *psxview* plugin was used to view processes in memory. This plugin can view hidden processes which is an advantage over the *pslist* plugin. All the process names that could not be accounted for from the Linux manual pages were noted with their process IDs (PIDs). The executable files in these memory locations were extracted to the analysis system using the *memdump* plugin. This plugin takes the PID as argument and a location on disk as destination to dump the memory image. The file types of the files dumped from the memory were checked with the Unix magic *file* utility and the files found to be ELF binary files were uploaded

to VirusTotal for scanning. The file samples identified as malicious after scanning by the anti-malware engines associated with VirusTotal were added to the sandbox testing pool.

3.4.2 Sandbox

In this section, a description of the sandboxes under evaluation is undertaken as well as the test setup procedure. These sandboxes are made up of a combination of open source tools and they serve as wrappers and front end to ease the analysis of the Linux malware samples.

3.4.2.1 REMnux

REMNux is a suite of tools implemented on Ubuntu Linux distribution for malware analysis. It has scripts and APIs to make requests to VirusTotal. Specifically, for Linux malware analysis, it has built-in tools such as *radare* (Radare, 2017) and GDB (Free Software Foundation Inc, 2017) for static analysis, *sysdiq* (Sysdig, 2017) and strace for dynamic analysis and *rekall* (Rekall Forensics, 2017) and *Volatility* for memory analysis. The suite of tools can be installed on an Ubuntu 14.04 system or used by downloading the complete virtual machine (REMnux Documentation Team, 2017). The latter approach was chosen because Ubuntu 14.04 is no longer the current Ubuntu LTS version (Canonical Ltd, 2017) and the installation script does not run on any other version of the distribution.

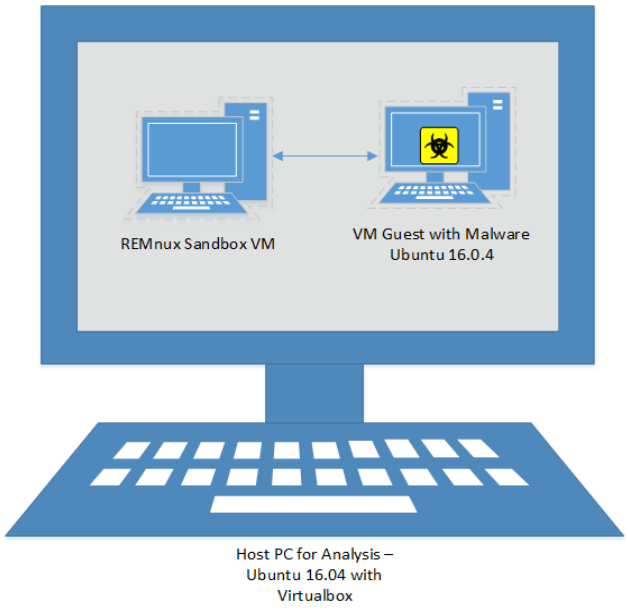


Figure 3. 3 Malware analysis with REMnux

Figure 3.3 is the setup for the analysis. The REMnux sandbox and the malware test host run as Virtualbox virtual machines on another Ubuntu 16.04 system. The REMnux sandbox emulates Internet and network services for the malware testing virtual machine using Inetsim (Hungenberg & Eckert, 2017) thus preventing the effects of the malware execution from leaving the confines of the Host PC. All static analysis tasks were performed on the REMnux sandbox virtual machine. Each sample from the repository were analysed using static, behavioural and memory analysis in turn. The samples and network traffic from the honeypot were also tested. Static analysis was performed by submitting the malware samples to VirusTotal using the *virustotal-search* utility. The malware detection engines

employed by VirusTotal are used to scan submitted samples with a resultant report indicating the number of engines that have detected the sample as malicious and the names of the engines as well as the name given the sample by the engines. The *r2* and *rabin* binaries from radare2 were used to investigate the ELF header files.

Dynamic analysis was performed by using the *strace* utility. The *strace* utility uses the *ptrace* system call to investigate the calls and library functions invoked by an program during execution. The arguments to these calls and functions are also available for analysis. This was done for each malware sample on the virtual machine guest for malware execution. A requirement of memory analysis is that a comparison be made between a clean snapshot of the virtual machine memory image and the memory snapshot after a malware sample has been executed. Using the Virtualbox virtual machine management utilities *debugvm* and *dumpvmcore,* the memory snapshot of the virtual machine was taken with the output in form of an ELF file. With *objdump* object file debugging utility being used to determine the main memory section memory base location and total offset, the system memory used by the virtual machine was located and extracted from the ELF file to a file format readable by *Volatility*. The foregoing process of memory extraction was repeated for each malware sample being tested. The malware samples were executed in turn on the guest virtual machine and the system memory extracted before being restored to the clean virtual machine state. The clean virtual machine guest image memory image was then compared with the memory image of the virtual machine after malware execution using the *linux_mem_diff.py* script on the REMnux virtual machine. This script runs *Volatility* memory analysis plugins against memory images and reports the differences.

3.4.2.2 Limon

Limon is a Python wrapper script that runs a sequential set of tests and procedures on a submitted malware sample. The tests involve static analysis of the malware sample, behaviour analysis and an option for memory analysis. It depends on the installation of Inetsim, *Yara rules*, *ssdeep*, Sysdig, Volatility, VMware workstation on the host machine. It uses Inetsim for network services emulation, Yara rules to determine if binaries are packed and the packing algorithm used. SSDEEP is used for fuzzy hashing calculation; this is used to determine the fuzzy hash output of the malware sample being examined. Sysdig is a front-end application for event monitoring and logging on the system. Volatility is a memory analysis library. The guests are implemented as VMware virtual machines which is the only virtualisation platform supported by Limon. Depending on the options chosen, different tasks are run. Some of the options are Internet mode in which the malware sample under examination is allowed unfettered access to the Internet. The alternative to Internet mode uses Inetsim to emulate network services. Limon uses Inetsim to emulate network services to avoid the need for the effect of malware sample execution to escape to the wider Internet. Other options are addition or exclusion of memory analysis. Limon uses Volatility for memory analysis.

Figure 3.4 is an illustration of the testing setup. Limon runs on the host system and the execution and analysis host is a VMware guest. Pre-execution and post-execution snapshots of the virtual machine guests are compared during the analysis. The author documentation (Monnappa K A, 2015) for setup proposed an Ubuntu 15.04 host system and Ubuntu 12.04 guest system. These operating systems were updated to the Ubuntu 16.04 for both the host and the guest system respectively in the testing environment.

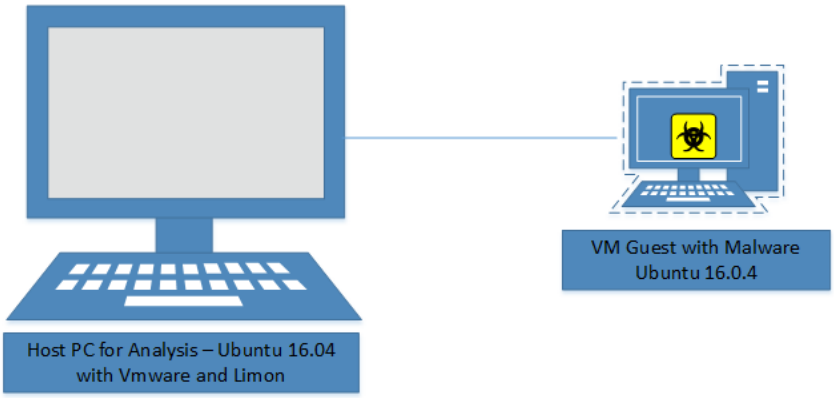


Figure 3. 4 Malware analysis with Limon

The guest virtual machine also requires installation of *Sysdig* and *strace* for event monitoring and dynamic analysis. A snapshot of the guest virtual machine in this state was created and the name of the snapshot added to the Limon configuration file on the host system. A Volatility image profile was also created. This is because memory analysis in Linux is dependent on the kernel version as memory arrangement and structure is operating system kernel dependent (Case & Richard, 2017).

When the Limon script is executed, some static analysis tasks are first performed on the malware sample (the name is part of the argument list during execution of the malware). An MD5 checksum of the malware sample is derived as well as a fuzzy hash function output. The MD5 checksum output is sent to VirusTotal to determine if the malware sample had been submitted to VirusTotal in the past for analysis. If the sample had been submitted to VirusTotal before, a scan test report is generated which shows the anti-malware engines (if any) that have successfully identified the malware sample as malicious and the name(s) it is known by. Limon also keeps a master list of fuzzy hash of all samples examined. This is used to compare similarity with the submitted samples.

For dynamic analysis, the malware sample is copied from the host system to the guest virtual machine and run with the strace utility to monitor the system calls. The sysdig utility is also used to monitor operating system events. If the option for memory analysis is selected, Volatility is used to compare the memory images of the virtual machine prior to execution and after execution. VMware stores memory images of virtual machines in vmem files (Aljaedi, Lindskog, Zavarsky, Ruhl, & Almari, 2011). These are examined by Volatility during the memory analysis.

The malware samples were placed in a directory and the Limon script was instructed to analyse all the samples in the directory with a run time of three minutes with the memory analysis option selected.

3.4.2.3 Cuckoo

Cuckoo is also a Python library that runs tests in sequence on submitted malware samples. It is integrated with open source analysis tools like Volatility, Yara, VirusTotal. It supports analysis of malware executed in guests implemented on VMware, Virtualbox and KVM. Cuckoo performs dynamic analysis by evaluating the behaviour of the sample during execution against its pool of 493 signatures which define triggers for different system and network activities. Figure 3.5 shows the analysis setup for Cuckoo.

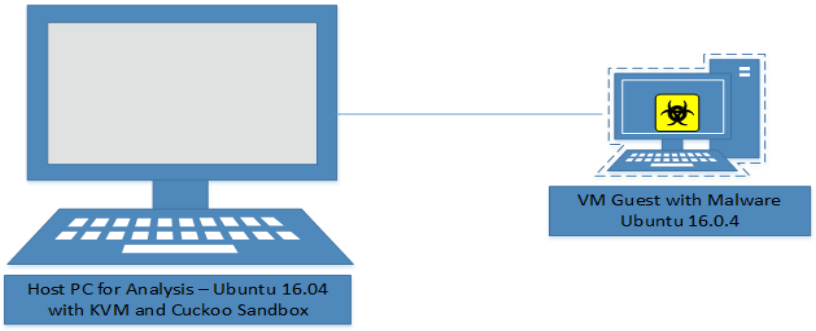


Figure 3. 5 Malware analysis with Cuckoo

Cuckoo testing setup is similar to Limon as it has the host operating systems running the sandbox software (Cuckoo) and a guest virtual machine to execute the malware samples. Cuckoo is a Python library that works with a combination of tools for malware analysis. Tools like Volatility, strace, and Yara are also used in Cuckoo. Cuckoo uses the *string* utility to investigate the string symbol table of ELF binaries as part of static analysis. Cuckoo supports VMware, Virtualbox, QEMU and KVM virtual machines. KVM was adopted for the analysis because of the performance benefits it offers as highlighted in (Bakhshayeshi, Akbari, & Javan, 2014; Younge et al., 2011). The virtual machine guest in Cuckoo depends on the inclusion and execution at system start-up of the *agent.py* script. This script orchestrates the analysis tasks on the guest. A *systemd* (freedesktop.org, 2017) unit file for the *agent.py* script was created as a service to ensure the script runs every time the system is started. The guest required the compilation and installation of a patched copy of *strace* binary for dynamic analysis to work. After the foregoing adjustments were made to the guest, a snapshot of the virtual machine was taken and the snapshot name was added to the Cuckoo configuration file on the host. This snapshot was used by Cuckoo to detect the effect of the execution of the malware samples by comparing the system state of the snapshot to the state of the virtual machine guests after the execution of each malware sample. The snapshot was also the state the virtual machine guest reverts to at the end of an analysis cycle.

For the analysis, Cuckoo was run in daemon mode and the malware samples were submitted as a batch job using the Cuckoo *submit.py* script. The Cuckoo configuration file was used to specify the options required such as the use of KVM, Inetsim to prevent malware interaction with the wider Internet and the need for memory analysis.

Figure 3.6 illustrates the setup for the malware analysis process using Detux. Detux supports analysis of Linux applications such as ELF binaries, scripts (Python, PHP, Perl and shell) written for x86, x86-64, MIPS, MIPSEL and ARM architectures. This is made possible by the provision of Debian Linux QEMU images in these architectures for the execution of the malware samples. Detux performs static analysis by using the *string*, *file* and *readelf* utilities. The *string* utility gives an output of all printable characters associated with the binary. The *file* program confirms the type of file being analysed detecting if it is an ELF binary or a script based on the file header information. If it is an ELF binary, it also determines the CPU architecture it was written for. The *readelf* program investigates the structure of the ELF file analysing the headers, determining the required libraries and confirming the intended CPU architecture.

3.4.2.4 Detux

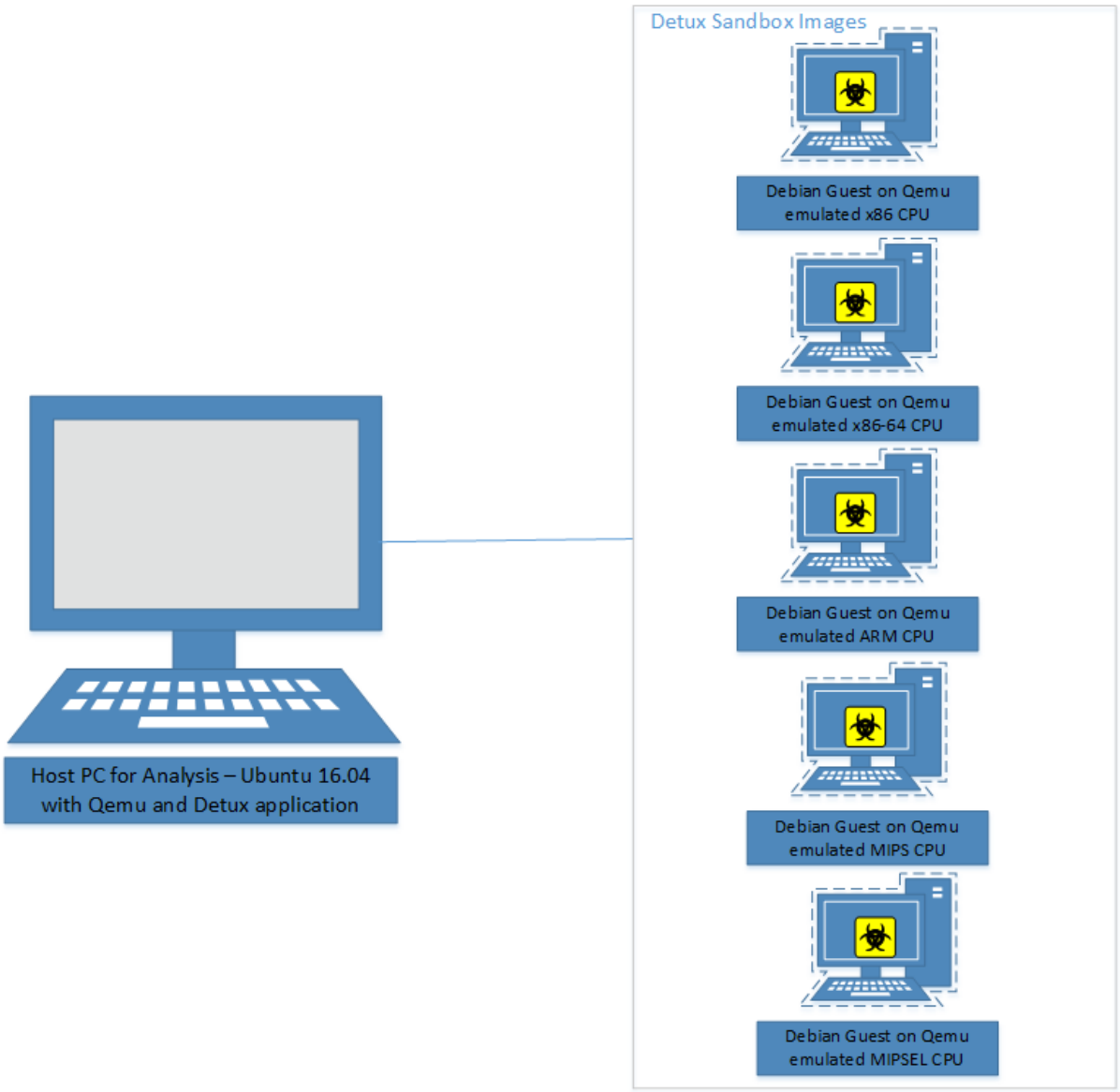


Figure 3. 6 Malware analysis with Detux

The dynamic analysis component of Detux requires that the QEMU virtual machine of the target CPU architecture be started. The target CPU architecture can be specified as an execution option or detected during the static analysis process. The malware sample is then copied from the host system to the running virtual machine and executed. During execution, the network interactions are observed by capturing the packets from the virtual network card of the host machine (Detux Sandbox, 2018).

As required by the setup instructions (Detux Sandbox, 2018), a virtual bridge was installed on the host systems and an IP address configured for it. The five QEMU images representing the supported CPU architectures were manually configured with IP addresses also. This was done to facilitate network communication and file transfer between the host analysis system and the virtual machines. The common bridge also allowed packet capture from the host system. Table 3.4 shows the IP and MAC

addressing allocation of the virtual machines. The host IP address was set as the default gateway and DNS server address of the virtual machines. *Inetsim* was executed on the host to emulate network services such as name resolution, network time synchronisation and Internet file transfer service interactions, preventing the effects of the sandbox execution from escaping to the wider Internet. The *detux.py* script was executed on the host for each malware sample and this initiated the file transfer to and subsequent execution on the appropriate guest virtual machine.

Table 3. 4 Virtual machines addressing information

Emulated CPU Type	IP Address (/24 mask)	MAC Address	Default Gateway/DNS Server
X86	10.180.1.2	00:11:22:33:44:51	10.180.1.1
X86-64	10.180.1.3	00:11:22:33:44:52	10.180.1.1
ARM	10.180.1.4	00:11:22:33:44:53	10.180.1.1
MIPS	10.180.1.5	00:11:22:33:44:54	10.180.1.1
MIPSEL	10.180.1.6	00:11:22:33:44:55	10.180.1.1

3.4.2.5 HaboMalhunter

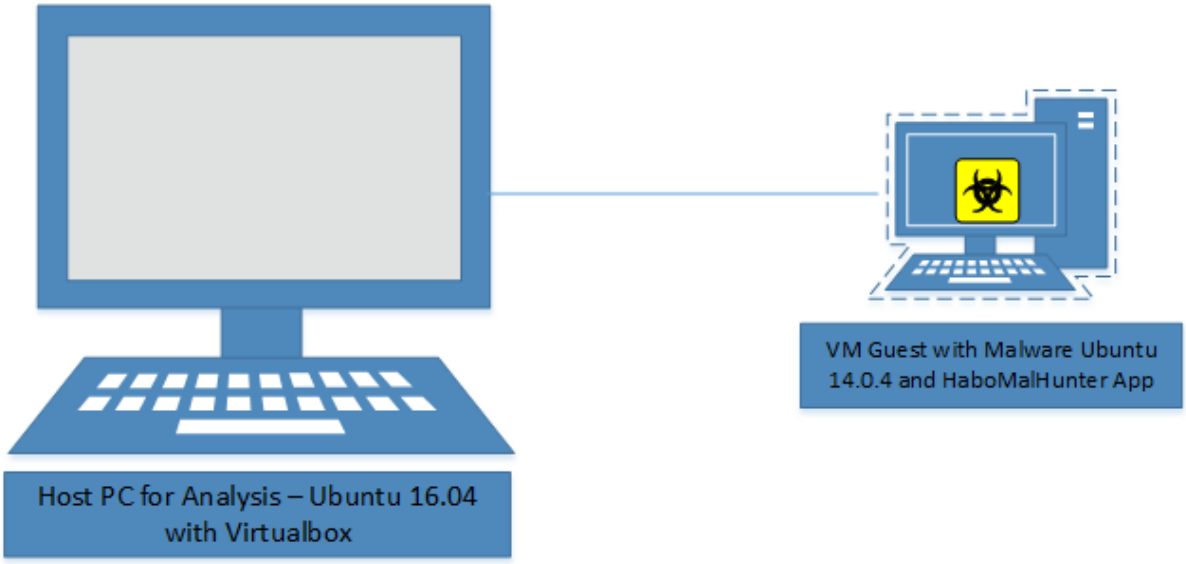


Figure 3. 7 Malware analysis with HaboMalHunter

HaboMalhunter also supports static analysis of malware samples using the *readelf* and *string* Unix utilities. HaboMalHunter uses a special program to load and execute ELF binary files instead of *strace* as employed by Limon. The process ID of this malware execution is then monitored by *Sysdig* and *tcpdump*. Sysdig monitors systems calls and file system activities while *tcpdump* monitors the network interactions.

As illustrated in figure 3.7, the malware sample is executed on a Virtualbox virtual machine guest. The host analysis system emulated network services for the guest using the *inetsim* Internet services emulation programme. The recommended virtual machine installation advises the use of the REMnux

virtual machine as base system for the installation. The Virtual machine was updated with the HaboMalhunter update scripts and a snapshot was taken. This snapshot was used as the execution environment for each malware sample analysis. Each sample was copied to the virtual machine. After execution and analysis of each malware sample, the reports generated were copied to the host system and the virtual machine state was restored to the saved snapshot for subsequent execution and analysis tasks.

3.5 Research Questions and Hypotheses

The object of the tests is to compare the evaluated malware sandboxes in features, effectiveness of analysis, reporting and resistance to obfuscation. The first hypothesis addresses the effectiveness of the open source analysis sandboxes. The first hypothesis, based on the description of the tools from the project pages and documentation mentioned in section 2.6 is:

Hypothesis H1: **The malware analysis sandboxes will collectively be able to detect indications of compromise from execution of all the malware samples**.

The second hypothesis that will be tested relates to the consistency of analysis results across the sandboxes.

Hypothesis H2: **All the analysis systems will have consistent analysis results for the malware samples executed and analysed**.

To address the hypotheses above, this research will answer the following sub-questions on the features of REMnux, Limon, Cuckoo, Detux and HaboMalHunter in the analysis of Linux malware samples.

3.5.1 REMnux

Does REMnux detect the presence of packing and the type of packing algorithm used?

Does REMnux detect metamorphic variants?

Is REMnux able to detect network, memory and operating system operations of malware samples after execution?

What batch processing and automated execution features are supported in REMnux?

What reporting features are available for malware analysis using REMnux?

3.5.2 Limon

Does Limon detect the presence of packing and the type of packing algorithm used?

Does Limon detect metamorphic variants?

Is Limon able to detect network, memory and operating system operations of malware samples after execution?

What batch processing and automated execution features are supported in Limon?

What reporting features are available for malware analysis using Limon?

3.5.3 Cuckoo

Does Cuckoo Sandbox detect the presence of packing and the type of packing algorithm used?

Does Cuckoo Sandbox detect metamorphic variants?

Is Cuckoo Sandbox able to detect network, memory and operating system operations of malware samples after execution?

What batch processing and automated execution features are supported in Cuckoo Sandbox?

What reporting features are available for malware analysis using Cuckoo Sandbox?

3.5.4 Detux

Does Detux Sandbox detect the presence of packing and the type of packing algorithm used?

Does Detux Sandbox detect metamorphic variants?

Is Detux Sandbox able to detect network, memory and operating system operations of malware samples after execution?

What batch processing and automated execution features are supported in Detux Sandbox?

What reporting features are available for malware analysis using Detux Sandbox?

3.5.5 HaboMalHunter

Does HaboMalhunter Sandbox detect the presence of packing and the type of packing algorithm used?

Does HaboMalhunter Sandbox detect metamorphic variants?

Is HaboMalhunter Sandbox able to detect network, memory and operating system operations of malware samples after execution?

What batch processing and automated execution features are supported in HaboMalhunter Sandbox?

What reporting features are available for malware analysis using HaboMalhunter Sandbox?

3.6 Conclusion

The research design was designed in this chapter drawing on the approaches used in similar work for sourcing and testing malware samples. The malware samples were obtained using a combination of the deployment of a honeypot and the use of online malware repositories. The discussion of the testing procedure was also undertaken and the research goals were presented in form of the research hypotheses and research sub-questions.

The result of the malware extraction from the honeypot and that derived from the execution of the malware samples in the evaluated sandbox environments are presented in chapter 4. The extraction process and output are described. The results of the analysis are organised by analysis type (static and dynamic) and automation and reporting features.

4. Results

4.1 Introduction

The research design and the reasons behind the research decisions were discussed in the previous chapter. The previous chapter addressed the sourcing of malware samples for testing as well as a description of the operation of the sandboxes and the steps for testing. This chapter presents the observations from the testing activities. The results of the honeypot analysis and investigation are highlighted in the following section. The third and fourth sections address the static and dynamic analysis outcomes respectively. The fifth section presents the results of the evaluation of the automation and reporting features of the sandboxes as well as the answers to the related research questions.

4.2 Honeypot analysis

Four (4) file samples out of the files carved out of the memory image of the honeypot system were identified as malicious when uploaded to VirusTotal. Figure 4.1 is a screenshot of the submissions to VirusTotal, showing the four files extracted from the memory image and the network traffic packet capture file.

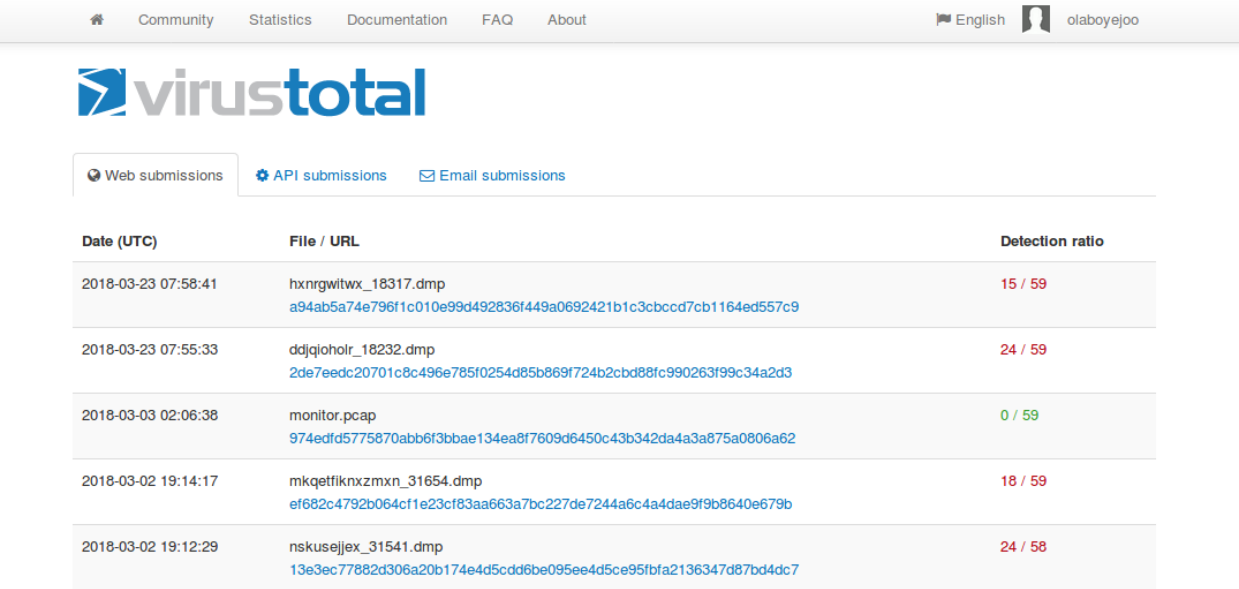
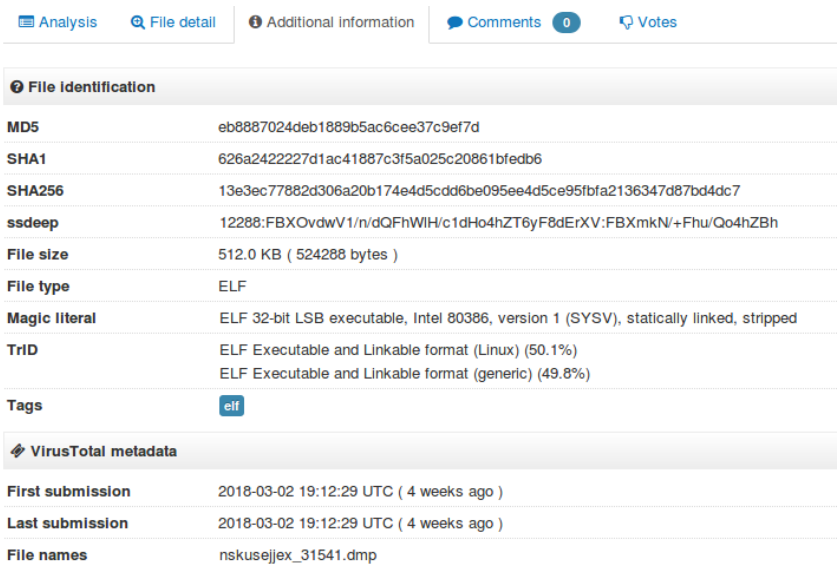


Figure 4. 1 Uploaded files to VirusTotal

The file names were automatically chosen for the extracted images by Rekall using the format – *process-name_process-id*.dmp where process-name and process-id were the process names and process identifiers from the process listings they were extracted from. *hxnrgwitwx_18317.dmp* and *ddjqioholr_18232.dmp* were uploaded on 23rd March 2018 and were flagged by fifteen (15) and twenty-four (24) out of fifty-nine (59) malware scanning engines as malicious respectively. *mkqetfiknxzmxn_31654.dmp* and *nskusejjex_31541.dmp* on the other hand were flagged as malicious

by eighteen (18) and twenty-four (24) malware scanning engines as malicious respectively. They were both uploaded on the 2nd of March 2018.

In line with the resolution in sub-section 3.3.2 about the acceptance of only malware samples that were first uploaded to VirusTotal in 2014 and beyond, the extracted samples were also examined for their date of first upload to VirusTotal. While the dates the files were uploaded are indicated in the foregoing paragraph, using the additional analysis option of VirusTotal, it is possible to make this confirmation using the SHA256 cryptographic hash values. This was in consideration of the possibility that the files have been uploaded previously under a different name.


Analysis | File detail | Additional information | Comments 0 | Votes

File identification

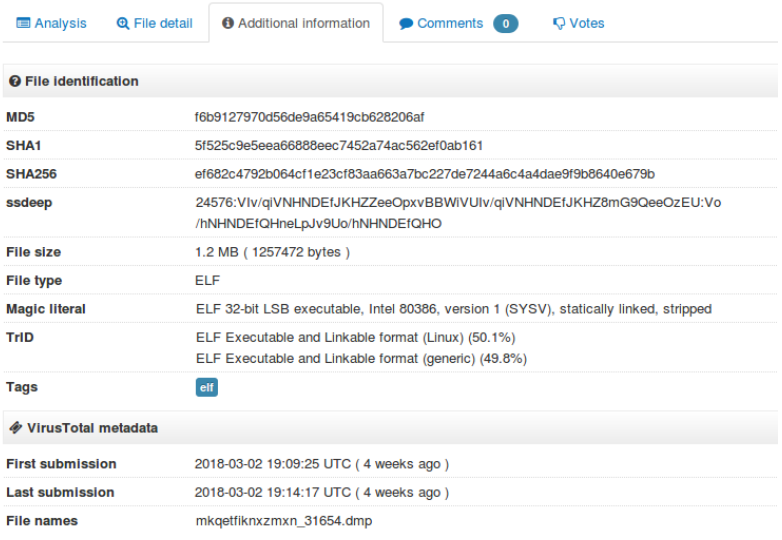
MD5	eb8887024deb1889b5ac6cee37c9ef7d
SHA1	626a2422227d1ac41887c3f5a025c20861bfedb6
SHA256	13e3ec77882d306a20b174e4d5cdd6be095ee4d5ce95fbfa2136347d87bd4dc7
ssdeep	12288:FBXOvdwV1/n/dQFhWlH/c1dHo4hZT6yF8dErXV:FBXmkN/+Fhu/Qo4hZBh
File size	512.0 KB (524288 bytes)
File type	ELF
Magic literal	ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, stripped
TrID	ELF Executable and Linkable format (Linux) (50.1%) ELF Executable and Linkable format (generic) (49.8%)
Tags	elf

VirusTotal metadata

First submission	2018-03-02 19:12:29 UTC (4 weeks ago)
Last submission	2018-03-02 19:12:29 UTC (4 weeks ago)
File names	nskusejjex_31541.dmp

Figure 4. 2 VirusTotal Analysis of nskusejjex_31541.dmp

Figure 4.2 is a screenshot of *nskusejjex_31541.dmp* with the file size displayed as 512 KB and first submission date displayed as the 2nd of March 2018.


Analysis | File detail | Additional information | Comments 0 | Votes

File identification

MD5	f6b9127970d56de9a65419cb628206af
SHA1	5f525c9e5eea66888eec7452a74ac562ef0ab161
SHA256	ef682c4792b064cf1e23cf83aa663a7bc227de7244a6c4a4dae9f9b8640e679b
ssdeep	24576:VIv/qiVNHNDEfJKHZZeeOpxvBBWiVUIv/qiVNHNDEfJKHZ8mG9QeeOzEU:Vo /hNHNDEfQHneLpJv9Uo/hNHNDEfQHO
File size	1.2 MB (1257472 bytes)
File type	ELF
Magic literal	ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, stripped
TrID	ELF Executable and Linkable format (Linux) (50.1%) ELF Executable and Linkable format (generic) (49.8%)
Tags	elf

VirusTotal metadata

First submission	2018-03-02 19:09:25 UTC (4 weeks ago)
Last submission	2018-03-02 19:14:17 UTC (4 weeks ago)
File names	mkqetfiknxzmxn_31654.dmp

Figure 4. 3 VirusTotal Analysis of mkqetifiknxzmxn_31654.dmp

Displayed in figure 4.3, *mkqetfiknxzmxn_31654.dmp* was first uploaded on the 2nd of March 2018. It is 1.2 MB in size.

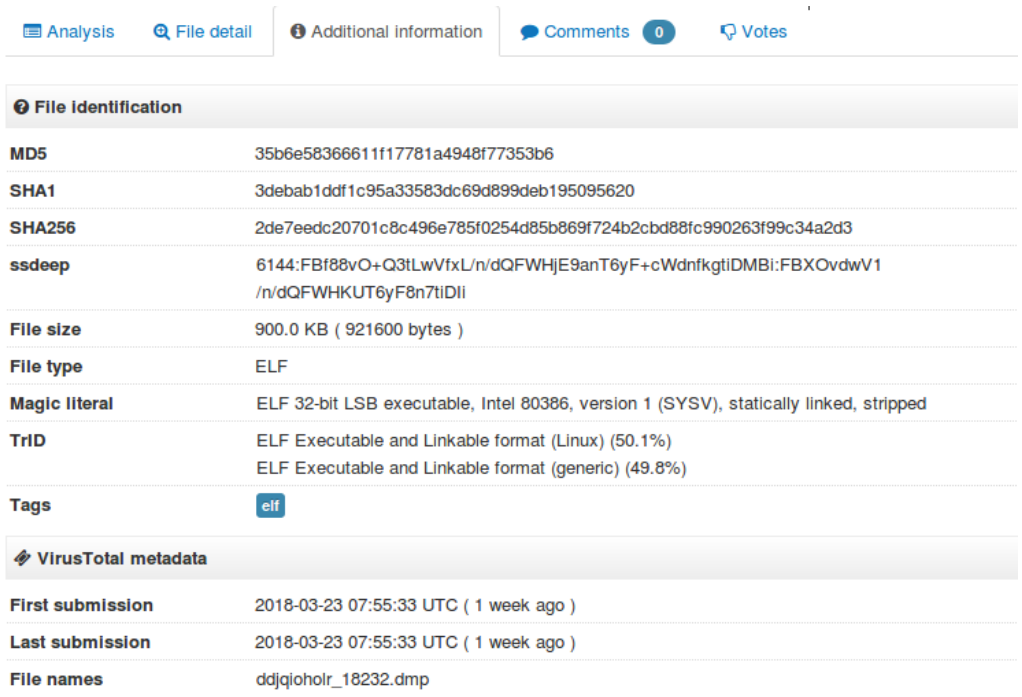


Figure 4. 4 VirusTotal Analysis of ddjqioholr_18232.dmp

In figure 4.4, a screenshot of *ddjqioholr_18232.dmp* is displayed. With a file size of 900 KB, it was first submitted to VirusTotal on 23rd of March 2018.

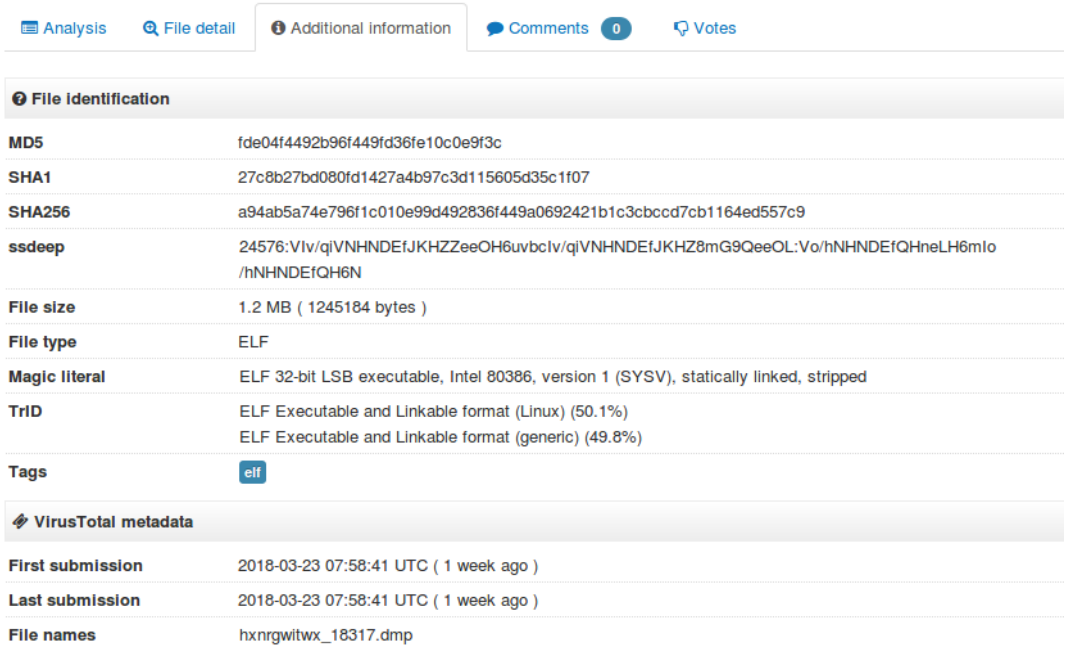


Figure 4. 5 VirusTotal Analysis of hxnrgwitwx_18317.dmp

Figure 4.5 shows a screenshot of the details of *hxnrgwitwx_18317.dmp.* It was first uploaded to VirusTotal on the 23rd of March 2018. It has a file size of 1.2 MB.

These first submission dates make these four samples eligible for addition to the testing pool because the dates are after 2014. The addition of the four extracted malware samples took the total number of malware samples to two hundred and ninety-seven (297). The network packet capture file was scanned using VirusTotal. The VirusTotal service utilises Snort and Suricata intrusion detection systems to scan packet capture files and the alerts triggered were reported.

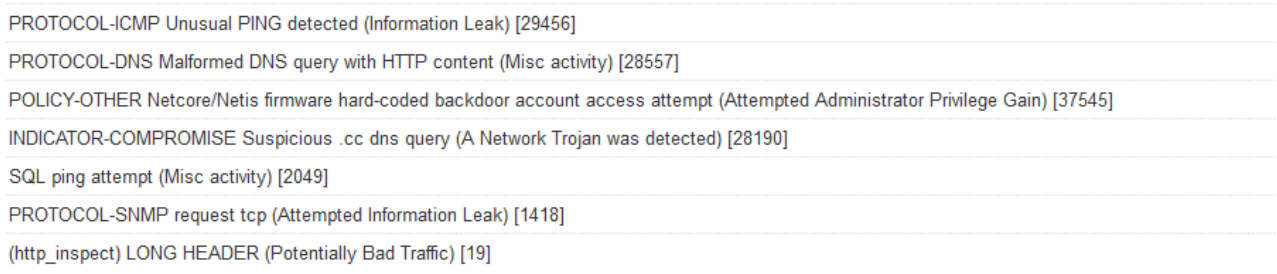


Figure 4. 6 Snort packet capture analysis screenshot

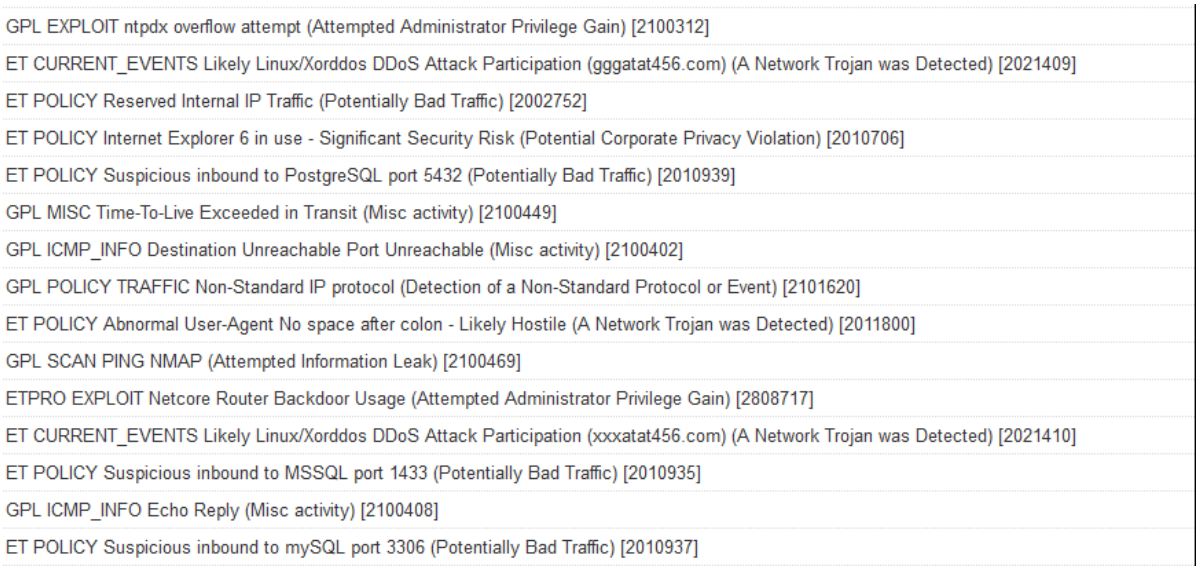


Figure 4. 7 Suricata packet capture analysis screenshot

Figures 4.6 and 4.7 are excerpts of the screenshot from the Snort and Suricata IDS analysis. Both reports alluded to the presence of a network trojan among other attack vectors such as signature alerts for DDoS participation, privileged escalation attempts and network reconnaissance.

4.3 Sandbox static analysis results

4.3.1 REMnux

The Radare2 reversing engineering binaries *rabin2* and *r2* were used for ELF header analysis and the investigation of dynamic loader references. The dynamic loader specified in the ELF header is

responsible for loading the program image to memory with the associated shared libraries required. Forty-five (45) of the malware samples requested the use of a dynamic loader, ten (10) of which requested the *ld-uClibc* dynamic loader preferred on embedded devices and resourced constrained systems (Cozzi et al., 2018). The examination of the ELF headers also revealed that six of the files have no section headers, thereby concealing the compile time view of the program from analysis. The ELF header also shows the entry points of programs; This is the virtual memory address that code execution begins at. On x86 and x86-64 CPU architectures, these entry points start at around the virtual memory addresses 0x8048000 and 0x400000 respectively (R. E. Bryant & O'Hallaron, 2015). Nineteen (19) of the malware samples were found to have entry points not in those regions. A possible reason for the use of different entry point regions is the use of packers. Packers select different entry point ranges to avoid conflict with the entry point the system program loader assigns the concealed binaries when they are eventually unpacked (Malin et al., 2013).

The VirusTotal query application component displayed the static analysis reports of the VirusTotal scanning. These results were the same as that derived from the portal indicating the malware analysis engines that have successfully identified the tested samples as malicious and the names associated with the samples.

4.3.2 Limon

The VirusTotal query aspect of the static analysis tasks resulted in same outcomes as that derived from the VirusTotal analysis report on each malware sample. The Linux *readelf* binary displayed the binary file program and section headers. The examination of the header files indicated that forty-five (45) of the binary samples requested the use of the dynamic linker, ten (10) of which were for the *ld-uClibc.so.0* library file while the remaining requests were for *ld-linux.so.2* and *ld-linux-so* library files. *ld-linux.so.2* and *ld-linux-so* usually refer to the same files using the soft link file references (Shotts Jr, 2012). The virtual memory address used as entry point of the all the 297 sample files were displayed. Nineteen (19) of the malware sample files had entry points that were a deviation from the ranges used in 32-bit and 64-bit Intel CPU architectures.

Limon made use of *ssdeep* fuzzy hashing utility to determine the degree of similarity between evaluated malware samples. When a sample is under examination, Limon stores it fuzzy hash value in a master file. Each examined sample's fuzzy hash is compared in similarity with the other entries in the master file to determine its similarity with every other entry. One hundred and eighty-seven (187) samples out of a total of two hundred and ninety-seven (297) samples in the testing pool were found to exhibit some degree of similarity with other samples. An illustration of the relationship between the degree of similarity as a percentage and the number of pairs found to display the level of similarity is shown in figure 4.8. The degree of similarity spanned a range from 25% to 100% with twenty-seven (27) pairs of samples observed to have 100% similarity by *ssdeep*. The most common extent of similarity was 91% displayed by two hundred and thirty-seven (237) pairs of malware samples.

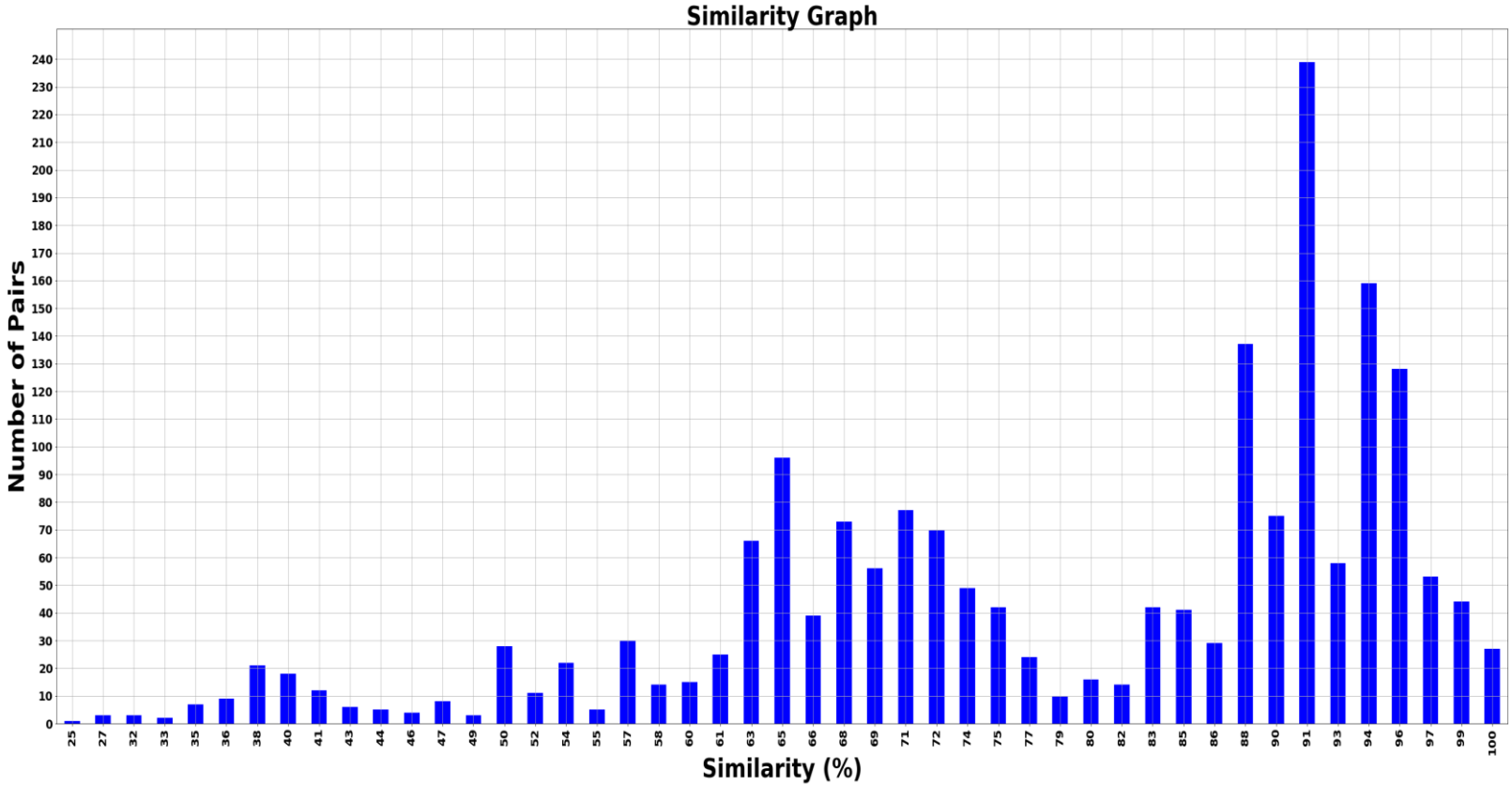


Figure 4. 8 Malware samples similarity graph using ssdeep

String analysis included references to packet flooding and encryption libraries, the use of the UPX packer, extraction of system settings and key strokes logging. Thirteen (13) of the malware samples had ASCII string references to commands that tried to extract the settings of the system. Another twenty-three (23) samples made references to altering the system. The methods referenced include encryption with text indicating the URL for decryption and changing the system start-up scripts or periodic execution (*cron* job) script to ensure a downloaded script runs every time the system starts or periodically. String analysis also revealed the packers in use in eight (8) malware samples. Eleven of the malware samples made references in the string symbol table to flooding and attack libraries and thirteen (13) samples made references to contacting control and command servers to download additional programs.

4.3.3 Cuckoo

The only static analysis performed by Cuckoo apart from the use of the *string* utility to view the string symbol table of the executable files was the call to the VirusTotal API and this returned the same output as that already derived from VirusTotal when the malware samples were extracted. The information indicated the malware scanning engines in use and the ones that have flagged the samples as malicious with the associated label given to the samples by those engines.

4.3.4 Detux

String analysis and ELF header analysis were performed by Detux during the static analysis phase. The ELF header analysis used the *readelf* utility and the results returned were the same as that reported by Limon with respect to the library call references and number of malware samples, whose entry point memory addresses varied from the traditional memory entry points for the process images executed on x86 and x86-64 CPU architectures. The string analysis using the *string* utility revealed text in seventy-eight (78) of the malware samples that gave indications of malicious intent. References to attempts at identifying the execution platform accounted for nineteen (19) of these samples. This was done by direct system settings extractions such as executing system configuration request commands and reading the system */proc* file system. Another method employed was examination of system error output when an invalid command is entered. The foregoing is a platform fingerprinting method employed by malware authors to eliminate platforms that are not the target for execution and it also serves as means for honeypot avoidance (Ullrich, 2016). Text references to directly alter the system by changing the system settings when the system is restarted, or direct alteration of system through encryption accounted for twelve (12) of the malware samples with indications of malicious intent. Other text references indicated the packer used in eight (8) samples. Ten (10) malware samples gave an indication of the class of devices that the malware was targeted at with the dynamic linker referenced in the string analysis output. The potentially harmful characteristics of the malware samples could be inferred by references to flooding protocol libraries and communications and file transfer activity with specific hosts in fourteen (14) and sixteen (16) of the malware samples respectively.

4.3.5 HaboMalHunter

Static analysis involved string analysis and ELF header analysis. The results derived for ELF header analysis are the same as that derived from Limon, REMnux and Detux. The string analysis results are

same as that derived from using Limon and Detux. The string analysis results are summarised below in table 4.1.

Table 4. 1 Summary of String and ELF header analysis

String references	Number of samples
UPX packer	8
Encryption libraries	3
Adjustments to start up or periodic scripts	9
Fingerprinting procedures	19
File transfer and communications with command centre	16
Traffic flooding attacks	14
/lib/ld-uClibc.so.0 dynamic loader	10

4.4 Sandbox dynamic analysis results

4.4.1 REMnux

One hundred and nine (109) of the malware samples could not be executed using *strace*. Ninety of them terminated execution by dumping their core files while the remaining nineteen (19) failed to run due to absence of requested libraries and unsuitability of the testing environment. The malware samples that ran successfully displayed indications of compromise by network connections to random hosts, connections attempt to botnet command and control servers and attempts to alter the system job scheduling file. One hundred and forty-nine (149) samples out of those that executed successfully were observed to be making connections to various hosts, twenty-nine (29) samples were attempting to reach command and control servers while three (3) samples were making changes to the system start up script.

4.4.2 Limon

The dynamic analysis results for Limon were similar to those derived from the REMnux analysis as *strace* was used in the dynamic execution analysis. Ninety (90) files terminated execution with a dump of the core while nineteen (19) malware samples did not execute all. Out of the latter, four failed to run due to lack of requested libraries, while two complained about the absence of AES-NI option on the CPU. From the samples that ran successfully, twenty-nine (29) were attempting to reach command and control servers, three (3) were making changes to the system boot scripts while One hundred and forty-nine (149) were attempting to connected to various IP addresses.

4.4.3 Cuckoo

Cuckoo relies on included signatures with internal thresholds for indications of compromise during dynamic analysis of malware sample. The VirusTotal signature was alerted for all the malware samples tested. Three of the samples raised alerts for the *suspicious_tld* and *network_http* signatures while fourteen samples raised alerts for the *network_icmp* signature.

4.4.4 Detux

Detux performed dynamic analysis by evaluating network traffic packet captures during malware execution. It was discovered that seventy (70) of the samples were attempting to make outgoing connections to various IP addresses. Thirty-six (36) malware samples were attempting to make connections to command and control servers to download scripts and replacement binaries for system binaries.

4.4.5 HaboMalhunter

From investigation of the network traffic during dynamic analysis, it was found that one hundred and twenty-one (121) files out of the malware samples attempted to contact various hosts on the wider Internet, six (6) malware samples attempted to connect to command and control servers while execution of one hundred and seventeen (117) samples showed traffic with loopback IP address (127.0.0.1) as source and destination address. It was also discovered during process analysis that the malware process exited with the segmentation fault exit code in forty-three (43) instances.

4.5 Automation and reporting features evaluation

4.5.1 REMnux

REMnux does not possess automatic virtual machine instrumentation so malware analysis cannot be automated because of the need to revert to a known clean snapshot after each analysis. The reporting features were limited to the output features of the individual scripts and utilities. The default console terminal standard output (*stdout*) of the tools (linux_mem_diff.py, strace, r2 and rabin) used by REMnux can be redirected to text files. Sysdiq also logs system interactions to text files. Unix text processing utilities like *grep*, *sed*, *awk* and the Python JSON Tool can be applied to manually format the text file outputs.

Sub Question 3.5.1d	*What batch processing and automated execution features are supported in REMnux?*
Batch processing was absent as it had no tools for virtual machine control.	
Sub Question 3.5.1e	*What reporting features are available for malware analysis using REMnux?*
Text output and the result of text processing tools.	

4.5.2 Limon

Automation was possible with Limon because the *limon.py* script handles the virtual machine orchestration allowing for a clean snapshot to be available for the analysis of each malware sample. The script also can be given a list of malware samples to analyse by taking advantage of the Unix command arguments expansion. In the tests undertaken, all the malware samples were place in a folder which was the command argument passed to the script. The limon.py script on completion of each malware sample analysis created a new folder for each of the malware samples tested. In each folder were text output of string and ELF header analysis, strace execution, memory analysis. A master file was kept for all the *ssdeep* operations which was used to test for the degree of familiarity with each individual file analysed.

Sub Question 3.5.2d	*What batch processing and automated execution features are supported in Limon?*
Batch processing was possible as the wrapper script controls the virtual machine and restores a clean snapshot for each execution run.	
Sub Question 3.5.2e	*What reporting features are available for malware analysis using Limon?*
Text output and the result of text processing tools.	

4.5.3 Cuckoo

Cuckoo was run in daemon mode so automation was possible by batch job submission using the *submit.py* script. The argument to the script was the directory where all the malware samples were stored. This also took advantage of the Unix command argument expansion. The server-side daemon on the analysis host and the *agent.py* script on the virtual machine worked in concert to ensure each analysis started with a clean snapshot. Cuckoo has a general text log file that reports on all the testing activities. Each analysed file also had a folder with the packet capture files and output reports. At conclusion of each malware analysis event, a JSON file and a HTML file were generated as reports.

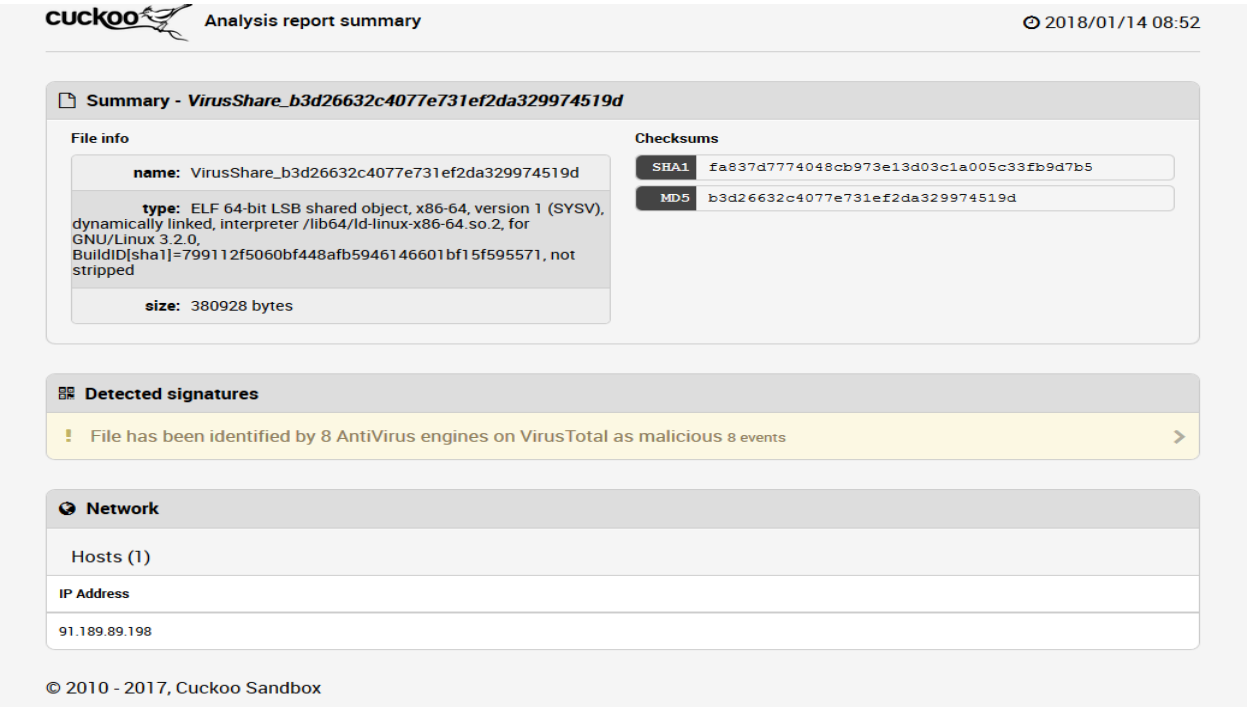


Figure 4. 9 Sample Cuckoo HTML report

Both reports classified the report into sections for VirusTotal reporting and specific Cuckoo signature analysis. Figure 4.9 is a screenshot of a sample Cuckoo HTML report. It shows the malware sample file information, the signatures that have detected the flagged the analysed sample as malicious and network communication activities of the virtual machine.

The sample JSON counterpart of the report is shown in figure 4.10. It contains the string analysis report in addition to the information contained in the HTML report. The screenshot below focuses on the string symbol analysis reporting section of the output.

```
    "2018-01-14 08:52:16,695 [cuckoo.core.plugins] DEBUG: Executed processing module \"Debug\" for task #1\n",
    "2018-01-14 08:52:16,704 [cuckoo.core.plugins] DEBUG: Running 473 signatures\n",
    "2018-01-14 08:52:17,066 [cuckoo.core.plugins] DEBUG: Analysis matched signature: antivirus_virustotal\n"
  ]
},
"strings": [
  "/lib64/ld-linux-x86-64.so.2",
  "libc.so.6",
  "__cxa_finalize",
  "__libc_start_main",
  "GLIBC_2.2.5",
  "_ITM_deregisterTMCloneTable",
  "__gmon_start__",
  "_ITM_registerTMCloneTable",
  "C:\\\\Users\\\\7\\\\Desktop\\\\dll - bak\\\\Release\\\\dll.pdbw",
  "TVT DEMOCONFIG-DESTORY\\\\\\\\\\.\\\\PIPE\\\\RUN_AT_SESSION (%d)",
  "writedroofiledroocmddrooHideUpdatePzfileHideSysUpfileHideSysCmdDesnation %s is small than finished!",
```


Figure 4. 10 Sample Cuckoo JSON report

Sub Question 3.5.3d	*What batch processing and automated execution features are supported in Cuckoo?*
Batch processing and automated analysis is possible. The combination of the backend daemon on the analysis host and the *agent.py* client script on the virtual machine organise the restoration to clean snapshot for each analysis.	
Sub Question 3.5.3e	*What reporting features are available for malware analysis using Cuckoo?*
HTML and JSON reporting are available.	

4.5.4 Detux

The detux.py script requires the specification of a destination report file for each file analysed. This made it impossible to use the Unix command argument expansion feature. The *find* command with the *exec* option was used to pass all the samples in a directory to the script file, allowing automatic analysis as well as the creation of unique output destination report for each sample.


```
{
    "cpu": "x86",
    "dns": [],
    "dns_request": [],
    "end_time": "2018-04-21T10:04:08.662204",
    "error": false,
    "filesize": 55840,
    "filetype": "Unknown",
    "interpreter": null,
    "ip": [
        "173.105.50.58",
        "20.129.172.37",
        "150.156.75.48",
        "172.179.70.59",
        "53.109.20.206",
        "152.240.207.169",
        "107.186.153.128",
        "121.241.44.157",
        "174.68.255.3",
        "158.228.49.67",
        "147.71.69.247",
        "122.227.237.216",
        "199.209.151.220",
        "104.113.145.62",
        "205.169.104.52",
        "191.237.69.127",
        "91.119.197.146",
        "122.199.62.14",
        "118.26.149.2",
        "171.0.22.231",
        "119.239.150.124",
        "104.0.187.141",
        "50.135.138.166",
        "90.171.75.210",
        "138.103.241.248",
        "113.159.135.224",
        "96.90.76.169",
        "77.64.120.115",
        "144.107.185.233",
        "61.186.220.71",
        "101.168.150.115",
        "137.79.76.239",
        "200.119.50.96",
```


Figure 4. 11 Sample Detux JSON report

A JSON file report was created for every sample analysed. The report separated the analysis reports into two categories, the network analysis section and the static analysis section that was further subdivided into the ELF header and string analysis sections. Figure 4.11 is an example of a Detux JSON report. The example report shows the network connections attempted following the execution of the malware sample.

Sub Question 3.5.4d	*What batch processing and automated execution features are supported in Detux?*
Automated analysis was possible with Unix find exec command option and the wrapper *detux.py* script that handled the start-up, shutdown and restoration of the virtual machine images.	
Sub Question 3.5.4e	*What reporting features are available for malware analysis using Detux?*
JSON reports	

4.5.5 HaboMalHunter

While analysis is initiated on the host system in Detux, Limon and Cuckoo, HaboMalHunter requires the analysis script to be executed from the virtual machine. This limits the amount of automation of analysis that can be possible because the virtual machine image must be restored to a vanilla snapshot state prior to execution and analysis of another malware sample. This is a similar situation to REMnux where analysis is also initiated on the virtual machine image. Where HaboMalhunter differs from REMnux is in the reporting. While HaboMalHunter creates unique string analysis, ELF header analysis, trace and memory dump log files for each malware samples prefixed with the sample name, the HTML and JSON reports used the generic names output.html and output.xpcn respectively. This necessitated the need to change the default names manually to match the sample names before being transferred to the analysis host. The report files aggregate the results for only the dynamic analysis activities with sections dedicated to network traffic analysis, file and process activities. Figures 4.12 and 4.13 show the JSON and HTML versions of the HaboMalHunter analysis report respectively.


```
Dynamic:
  Process:
    Execute a file:   []
    fork:
      0:              "fork ret=31833, args=[] "
  Net:
    TCP:
      0:              "192.168.56.101 -> 73.135.11.121 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      1:              "192.168.56.101 -> 197.150.133.186 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      2:              "192.168.56.101 -> 197.148.213.115 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      3:              "192.168.56.101 -> 38.209.169.32 TCP 56 46863 -> 2323 [SYN] Seq=0 Win=42271 Len=0"
      4:              "192.168.56.101 -> 85.164.186.188 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      5:              "192.168.56.101 -> 89.165.38.106 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      6:              "192.168.56.101 -> 141.49.72.74 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      7:              "192.168.56.101 -> 46.23.91.137 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      8:              "192.168.56.101 -> 4.28.173.22  TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      9:              "192.168.56.101 -> 175.244.59.79 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      10:             "192.168.56.101 -> 103.20.198.240 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      11:             "192.168.56.101 -> 205.250.44.173 TCP 56 46863 -> 2323 [SYN] Seq=0 Win=42271 Len=0"
      12:             "192.168.56.101 -> 195.175.95.14 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      13:             "192.168.56.101 -> 143.37.76.201 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
      14:             "192.168.56.101 -> 89.64.203.118 TCP 56 46863 -> 23 [SYN] Seq=0 Win=42271 Len=0"
    UDP:              []
    DNS:              []
  Other:              {}
  File:
    File read:
      0:              "read: path=/lib/x86_64-linux-gnu/libm.so.6, size=832"
      1:              "read: path=/lib/x86_64-linux-gnu/libc.so.6, size=832"
      2:              "read: path=/lib/x86_64-linux-gnu/libpthread.so.0, size=832"
    Lock file:        []
    File open:
      0:              "open: path=/lib/x86_64-linux-gnu/libpthread.so.0, flags=O_RDONLY|O_CLOEXEC, mode=0"
      1:              "open: path=/lib/x86_64-linux-gnu/libm.so.6, flags=O_RDONLY|O_CLOEXEC, mode=0"
      2:              "open: path=/lib/x86_64-linux-gnu/libc.so.6, flags=O_RDONLY|O_CLOEXEC, mode=0"
      3:              "open: path=/lib/x86_64-linux-gnu/librt.so.1, flags=O_RDONLY|O_CLOEXEC, mode=0"
```


Figure 4. 12 Sample HaboMalHunter JSON report

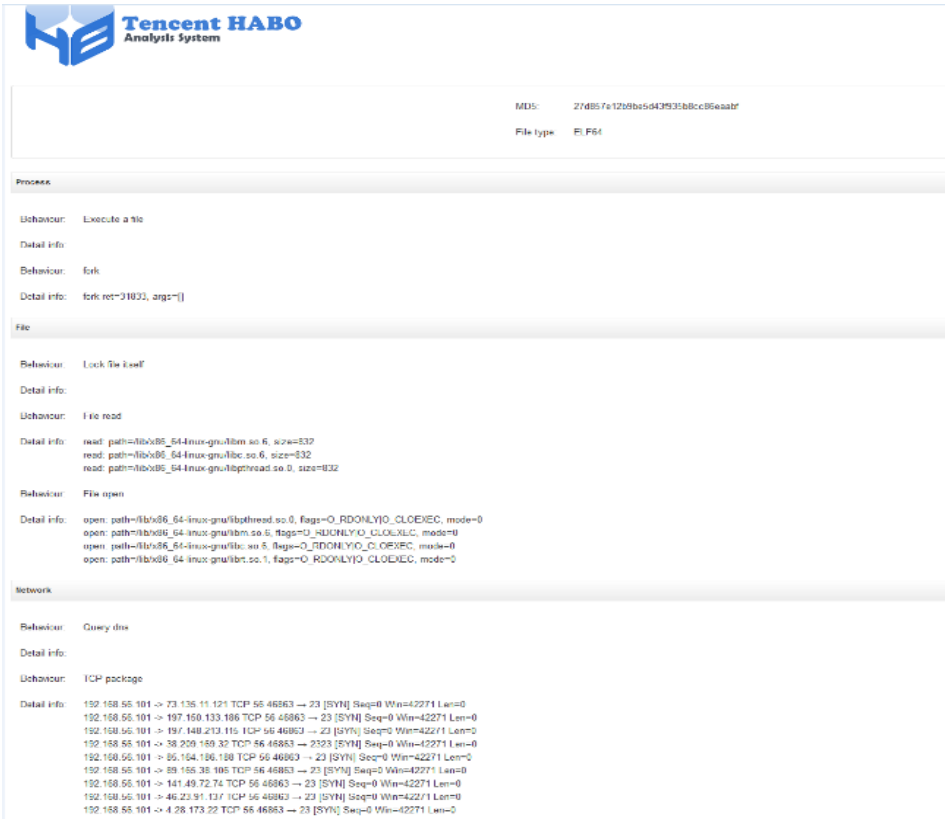


Figure 4. 13 Sample HaboMalHunter HTML report

Sub Question 3.5.5d	*What batch processing and automated execution features are supported in HaboMalHunter?*
HaboMalHunter is unable to support batch processing of malware samples because like REMnux, it does not have a method to control the virtual machine.	
Sub Question 3.5.5e	*What reporting features are available for malware analysis using HaboMalHunter?*
HTML and JSON reports are available.	

4.6 Conclusion

The result of the testing activities was presented and organised by analysis type for each sandbox tested. The reporting and automation features were also discussed with screenshots and descriptions of reports for sandboxes that offered more than text file output. The research sub-questions on automation and reporting were also answered. The following chapter is a discussion of the test results in the context of the nature of samples in the dataset and the conclusions that can be inferred about the relative effectiveness of the sandboxes.

5. Discussion

5.1 Introduction

The previous chapter presented the results of the testing and analysis of the malware samples on the different sandbox environments. This chapter is a discussion of the results considering the characteristics of the malware samples used in the dataset. The next section breaks down the malware samples by CPU architecture and malware families. The characteristics of the malware families are also summarised to correlate with the analysis results. The following sections on static and dynamic analysis discusses the results with respect to these classifications of the dataset. The final section in this chapter presents the answers to the research questions and hypotheses in section 3.5.

5.2 Dataset family classification

The malware samples in the testing pool consists of binaries targeted at the x86 and x86-64 CPU architectures. There were two hundred and seventy-five (275) and twenty-two (22) samples respectively in the dataset. The AVCLASS program was used to label the malware samples and indicate the families they belong.

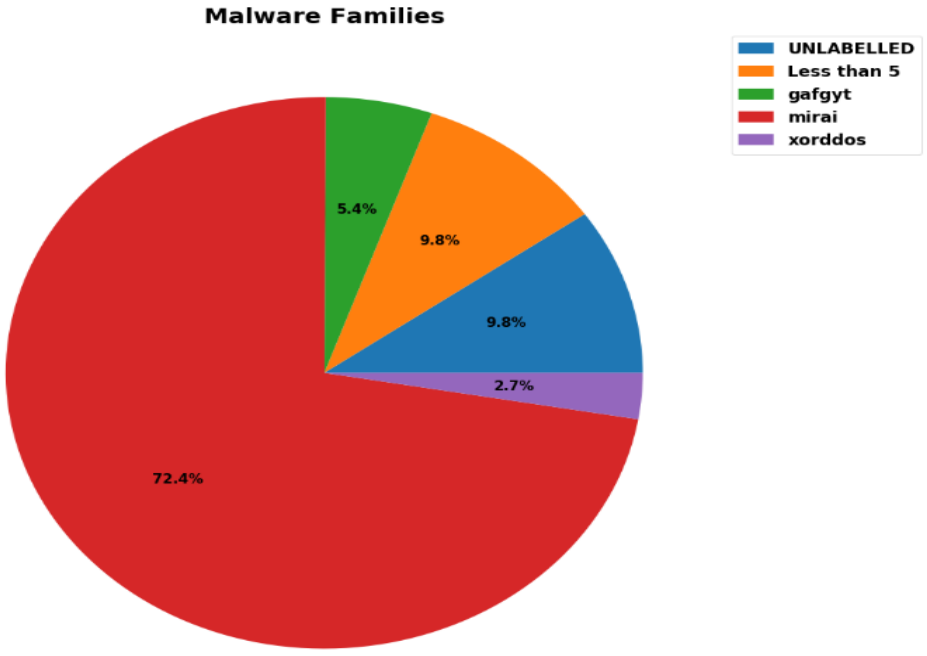


Figure 5. 1 Testing pool malware classification

As shown in Fig 5.1, Most of the malware samples are from the *Mirai* family. Two hundred and fifteen (215) of the malware samples were classified as part of the *Mirai* family. There were sixteen (16) malware samples in the testing pool from the *Gafgyt* family. The *Xorddos* family make up the last major family of malware in the pool that was labelled by AVCLASS. The class had eight (8) malware samples. Twenty-nine (29) of the samples could not be labelled and the remaining twenty-nine (29)

samples consisted of malware samples in families with less than five members. Table 5.1 is a summary of the characteristic of the malware families identified in the testing dataset.

Table 5. 1 Summary of Malware family

Family	Number of Samples	Summary of characteristics
Adore	1	Downloads binaries from remote hosts and replaces systems *ps* utility with a modified version. Steals information from host and seeks new systems to infect (Symantec, 2018b)
Binom	1	Primarily a dropper file that infects other ELF files and they in turn seek out other ELF files to infect (McAfee, 2018)
Bonk	1	Used to install malware remotely on compromised system. (Microsoft, 2018b)
Ddostf	1	Backdoor exploit that downloads other malware, makes remote access connections, captures keyboard strokes and used for DDoS attacks (Fortiguard, 2018a)
Dofloo	2	Opens backdoor on compromised system. Modifies boot up script so it starts everything system is restarted and used for DDoS attacks (Symantec, 2018c)
Gafgyt	16	Backdoor connection to command centre where it receives commands to effect DDoS and information theft (Symantec, 2018d)
Erebus	1	Ransomware that bypasses user access control and starts encrypting files with specific suffixes (Redhat, 2018)
Grip	1	Infects legitimate programs and stops them from working (Microsoft, 2018f)
Local	1	Exploit on compromised system that allows installation of other programs (Microsoft, 2018c)
Mibsun	1	Backdoor trojan that allows compromised system to be controlled remotely, also participates in information theft from compromises system (Fortiguard, 2018b)
Midav	1	A tool to spoof network addresses with logic to adapt to the network configuration of compromised system and allows system to be controlled remotely (Microsoft, 2018e)
Miner	2	Attacker connects to victim via brute-force attack then installs miner trojan to use system CPU to mine Monero (XMR) cryptocurrency. (Dr Web, 2018)
Mirai	215	Exploits Universal Plug and Play vulnerability to downloads a script from remote server and launch DDoS attacks (Symantec, 2018e)
Mumblehard	1	Connects to remote locations to download files. Turns compromised host to a spam bot (Symantec, 2018f)
Nestea	1	Exploits IP fragmentation vulnerability on Linux 2 kernel to cause DOS attacks on host system (Insecure, 2018)
Pnscan	1	Brute-force SSH attack on a system. Victim system is then used to attack other systems with SYN flood (NJCCIC, 2018)
Race	1	Exploits system vulnerability to install other malware (Microsoft, 2018d)
Scalper	1	An Apache vulnerability exploit to cause denial of service on victim host (F-Secure, 2018)
Setag	3	Leaves victim open to unauthorised access through a backdoor connection (Microsoft, 2018a)
Snoopy	1	File infector, monitors process when file is launched (Trend Micro, 2018c)
Sshgo	2	Connects to systems with weak authentication. Connects to command and control server downloads malware to scan for others with weak authentication credentials (Talos, 2018)
Svat	1	Overwrites standard library on Linux systems with downloaded copy from remote host (Trend Micro, 2018a)
Thou	1	Infects legitimate programs and stops them from running (Microsoft, 2018g)
Tsunami	1	Communicates with Internet Relay Chat (IRC) command and control server. It receives commands to launch DDoS attacks on other hosts (Monnappa, 2015)
Turla	1	Synchronizes files system between compromised hosts and remote host using remote backdoor connection (Symantec, 2018g)
Xorddos	8	Downloads scripts from command centre and ensures persistence by altering system start up script. It conceals its activities by installing a rootkit. It partcipates in DDoS attacks (Symantec, 2018h)
Znaich	1	Downloads script from command and control centre and participates in DDoS attack. It Installs a rootkit to hide network and file ensuring persistence with a *cron* job at system start up (Trend Micro, 2018b)

The characteristics of the malware samples include participation in distributed denial of service attacks under the control of a command centre, ransomware and bitcoin mining and information theft. The Xorddos, Mirai, Gafgyt are some of the families used for launching DDoS attacks. Erebus and its variants are a family of ransomware. The Miner family enlists the compromised system as a cryptographic currency mining bot.

5.3 Static Analysis

This section discusses the results of the static analysis. These results are discussed with reference to the underlying tools employed by the sandboxes and the combination of the results obtained from the different tools with respect to the malware classes. The sub-questions related to static analysis are also answered in this section.

5.3.1 Obfuscation and packing

Limon was unique in its use of *ssdeep* to detect metamorphic variations of the malware samples. There were 187 malware samples in 1913 pair combinations with varying degrees of similarity from 25% to 100%. Some of the later discussions will be placed in the context of this similarity information especially in cases where malware samples were unlabelled according to families. Figure 5.2 shows the comparison of malware families for the sample pairs that have degrees of similarity. For pairs of malware samples with 100% similarity, twenty-three (23) out of twenty-seven (27) malware samples are members of the same malware family as classified by AVCLASS. Thirty-nine (39) out of forty-four (44) malware samples with 99% have the same family memberships. Forty-two (42) out of fifty-three (53) malware samples with 98% similarity are of the same family. Malware sample pairs with 86%, 88%, 91% similarity have twenty-four (24) out of twenty-nine (29), one hundred (100) out of one hundred and thirty-nine (139), one hundred and eighty-six (186) out of two hundred and thirty-seven (237) as members of the same malware family respectively.

The *readelf* binary is part of the GNU Binary Utilities *binutils* package (Sourceware, 2018a). It was used as part of the static analysis component of Limon, HaboMalhunter and Detux sandboxes. The result of the ELF header analysis performed by *readelf* was the same for the three sandboxes. Radare2 was used for ELF header analysis by REMnux and its results were similar as those derived from the sandboxes that employed *readelf*. Using both *radare2* and *readelf* revealed that nineteen (19) of the malware samples have memory entry points that are different from the default range used by x86 and x86-64 CPU architectures.

The string analysis detected references to the UPX packer in eight (8) of the malware samples. Seven (7) of them were among the nineteen found to have memory entry points that were not within the regular ranges for the CPU architectures used. String analysis using the *string* utility was used by Limon, Cuckoo and HaboMalhunter. Eight (8) of the sixteen (16) malware samples that had string references to connections to a remote command and control centre for commands and file downloads were found to be in the Mirai family while the other eight were labelled as Gafgyt. This is consistent with the behaviour of both families as listed in table 5.1. Out of the nineteen (19) files that employed host fingerprinting procedures, two (2) had the family name of Setag while fifteen (15) had the family name of Mirai. Out of the two that were unlabelled, taking *ssdeep* similarity index into account, one of them had 94% similarity with another sample labelled as part of the Mirai family.

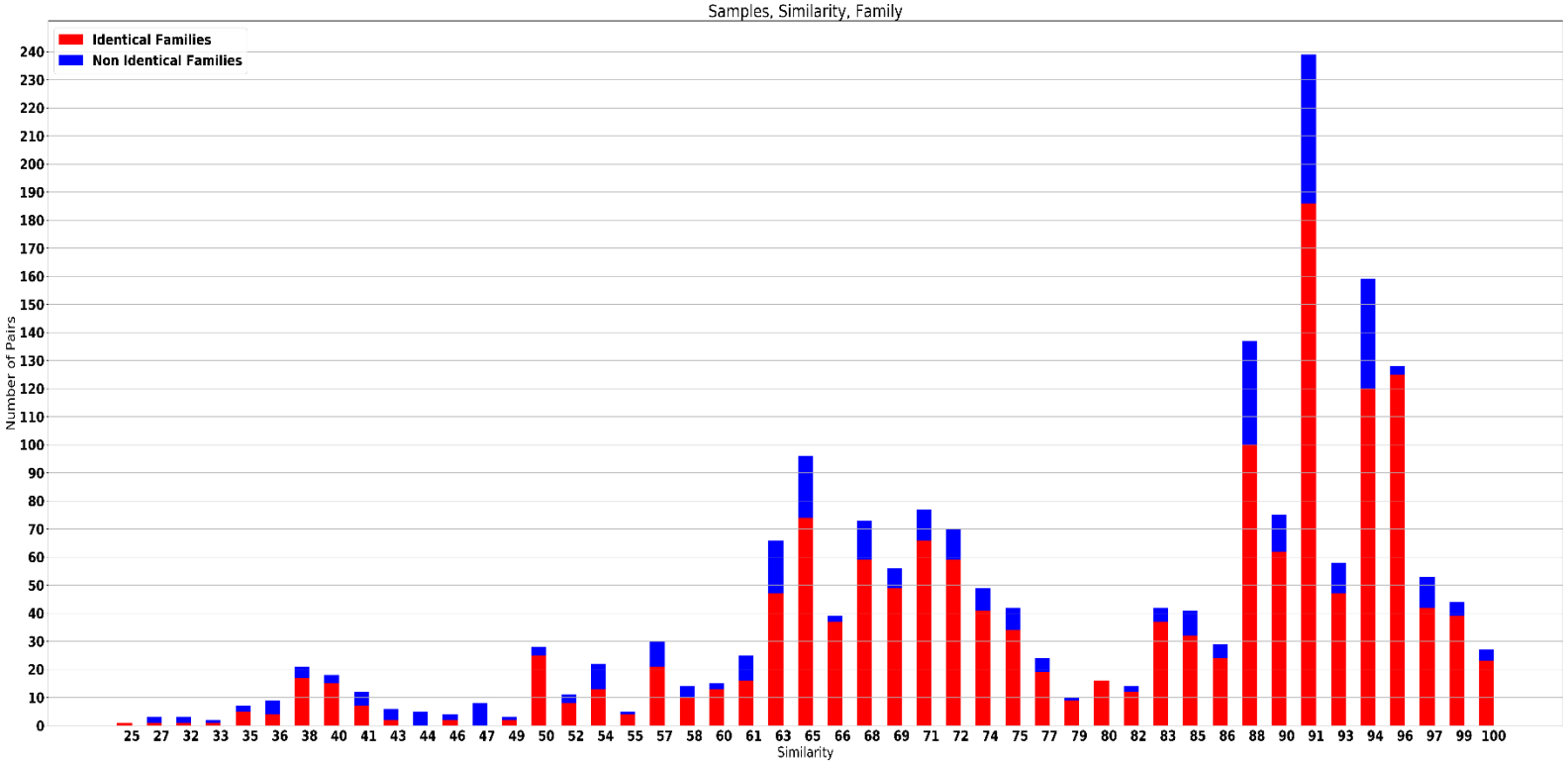


Figure 5. 2 Malware sample similarity and malware family comparison

Two of the three files with string references to encryption were unlabelled while the last one was categorised as belonging to the Erebus family using AVCLASS. The Erebus class as noted in Table 5.1 is a family of ransomware.

Limon in addition to ELF header analysis and string symbol table analysis also used *Yara rules* to detect the packer used in four of the malware samples, two of which were among the nineteen detected by *readelf* to be using different memory entry points from that associated the x86 and x86-64 CPU architectures. Cuckoo also supports the use of *Yara rules;* however, in the tests, there was no indication of any detected packers.

5.3.2 Virustotal

Limon and Cuckoo employed the VirusTotal API in static analysis by submitting the MD5 hash function output of the malware samples to VirusTotal. The resultant report was the same as static call to VirusTotal. The results showed the anti-malware engines that have successfully identified the samples as malicious and the given names by the engines. This method of static analysis is only effective for samples that have been previously submitted to VirusTotal for analysis. REMnux's application of VirusTotal submits the sample to VirusTotal for analysis. This could be useful for samples that have yet to be submitted to VirusTotal. The utility of this could not be confirmed as all the samples used in the experiment have been previously scanned by VirusTotal with its participating engines.

5.3.3 Answers to sub-questions on packing and obfuscation

Sub Question 3.5.1a	*Does REMnux detect the presence of packing and the type of packing algorithm used?*
REMnux detects the presence of packing using *radare* if the entry point memory address is not that associated with the CPU architecture.	
Sub Question 3.5.1b	*Does REMnux detect metamorphic variants?*
REMnux does not detect samples that are metamorphic variants	

Sub Question 3.5.2a	*Does Limon detect the presence of packing and the type of packing algorithm used?*
Limon also detects the presence of packing using *readelf* if the entry point memory address is not that associated with the CPU architecture. Limon, using Yara rules, detects the type of packing algorithm employed depending on the installed ruleset. The string analysis can also be used to determine the type of packer used.	
Sub Question 3.5.2b	*Does Limon detect metamorphic variants?*
Limon's use of storing ssdeep fuzzy hash outputs of analysed samples allows it to determine similarities in file structure that makes it effective for detecting metamorphic variants.	

Sub Question 3.5.3a	*Does Cuckoo detect the presence of packing and the type of packing algorithm used?*
Cuckoo can detect the presence of packing using string analysis as this gives an indication of the packers used when added to the string symbol table.	
Sub Question 3.5.3b	*Does Cuckoo detect metamorphic variants?*
Cuckoo does not detect samples that are metamorphic variants.	

Sub Question 3.5.4a	*Does Detux detect the presence of packing and the type of packing algorithm used?*
Detux can detect the presence of packing using string analysis and readelf as the former reads the string symbol table for references to known packers and the latter detects packing by observation of the virtual memory entry points.	
Sub Question 3.5.4b	*Does Detux detect metamorphic variants?*
Detux does not detect samples that are metamorphic variants.	

Sub Question 3.5.5a	*Does HalMalHunter detect the presence of packing and the type of packing algorithm used?*
The use of readelf and string analysis allows HalMalHunter to detect the presence of packing and the packer employed. With the former, the memory entry point of the binary is the indicator. The latter relies on the string symbol table references to known packers.	
Sub Question 3.5.5b	*Does HalMalHunter detect metamorphic variants?*
HaboMalhunter does not detect samples that are metamorphic variants.	

5.4 Dynamic Analysis

The dynamic analysis results for the different sandboxes are discussed in this section. The relationship between the indications of compromise and malware family classification are examined. The Limon ssdeep similarity list was used to infer the families of unlabelled malware samples where possible. This discussion is also extended to the CPU families. The research sub questions related to dynamic analysis are also addressed.

5.4.1 REMnux

The choice of tools used in REMNux were discretionary based on the recommendations in the documentation for tools available for static and dynamic analysis on Linux systems as it does not possess an automatic analysis application or wrapper like the other sandboxes tested.

A total of One hundred and forty-nine (149) malware samples were observed attempting to connecting to multiple hosts. One hundred and thirty-eight (138) of these were of the Mirai family, four (4) were from the Gafgyt family and one each from the SSHgo and Pnscan families. Five of the malware samples were unlabelled.

The other major category of indication of compromise is the attempt by some malware samples to connect to command and control servers to download files, creating remote backdoor connections to

facilitate reception of commands. There were twenty-nine (29) malware samples in this category. Twelve (12) of these were of the Gafgyt family, nine (9) were from the Mirai family, four (4) were from the Xorddos family while Setag, Mumblehard, Turla and an unlabelled sample each contributed one sample each.

Sub Question 3.5.1c	*Is REMnux able to detect network, memory and operating system operations of malware samples after execution?*
REMnux was able to detect indicators of compromise in 181 out of the 297 malware samples across the two CPU architectures.	

5.4.2 Limon

Limon recorded similar results as REMnux since its dynamic analysis is also based on *strace*. The major categories of indications of compromise are the observation of malware samples attempting to connect to various hosts and attempts to connect to specific hosts. Limon reported the same results as REMnux in both categories. Table 5.2 shows the distribution by family of the malware samples attempting to connect to various hosts on the wider Internet. Limon also recorded indications of compromise for thirteen (13) of the twenty-two (22) malware samples built for x86-64 processors and one hundred and sixty-eight (168) out of two hundred and seventy-five (275) malware samples built for the x86 architecture.

Table 5. 2 Malware samples making random connections by family (Limon)

Family	Number of Samples
Mirai	138
Unlabeled	5
Gafgyt	4
SSHgo	1
PnScan	1

Sub Question 3.5.2c	*Is Limon able to detect network, memory and operating system operations of malware samples after execution?*
Limon detected indicators of compromise in 181 out of the 297 malware samples across the two CPU architectures.	

5.4.3 Cuckoo

The indications of compromise detected by Cuckoo were in form of custom signatures alerts during analysis. The custom signatures flagged were *suspicious_tld*, *network_icmp* and *network_http*. Three (3) samples on execution displayed characteristics that matched the *suspicious_tld* signature and all three were of the Mirai family. Thirteen (13) malware samples from the Mirai family and one unlabelled sample triggered the network_icmp signature. Three malware samples made up of two from the Xorddos and one from the Mumblehard family were flagged by the *network_http* signature. Table 5.3 is a breakdown of the signature alerts triggered by malware family.

Table 5. 3 Triggered signatures by family (Cuckoo)

Signature	Family	Number of Samples
network_icmp	Mirai	13
	Unlabeled	1
suspicious_tld	Mirai	3
network_http	Xorddos	2
	Mumblehard	1

There were eighteen (18) unique samples for which signatures were triggered during dynamic analysis and they were all built for the x86 CPU architecture.

Sub Question 3.5.3c	*Is Cuckoo able to detect network, memory and operating system operations of malware samples after execution?*
Cuckoo detected indications of compromise on 18 out of 297 malware samples and they were all on the x86 CPU architecture.	

5.4.4 Detux

The only two categories of indications of compromise discovered using Detux were attempts by malware samples to connect to command and control centres and attempts to connect to multiple hosts on the wider Internet. Table 5.4 shows the distribution of the latter class by malware families. Sixty-two (62) of the seventy (70) malware samples attempting to connect to multiple hosts were of the Mirai family. The Gafgyt family accounted for three (3) of the malware samples with SSHgo accounting for just one of the malware samples. Four (4) of the malware samples were unlabelled with two of them having similarity of 90% (using *ssdeep* context based hashing results from Limon) with samples in the Mirai family. One of the unlabelled samples also recorded 99% similarity with a sample in the Mirai family.

Table 5. 4 Malware samples connecting to multiple hosts by family (Detux)

Family	Number of Samples
Mirai	62
Gafgyt	3
SSHgo	1
Unlabelled	4

Table 5.5 shows the malware family distribution of malware samples attempting to connect to specific IP addresses for command and control instructions and scripts. Out of the thirty-six (36) samples in this category, twenty (20) were from the Mirai family. The other major categories were Gafgyt and Xorddos which recorded six (6) and four (4) samples respectively.

All one hundred and six (106) malware samples that displayed indications of compromise during Detux dynamic analysis were built for the x86 CPU architecture.

Table 5. 5 Malware samples connecting to control centres by family (Detux)

Family	Number of Samples
Mirai	20
Gafgyt	6
Xorddos	4
Dofloo	2
Mumblehard	1
Turla	1
Setag	1
Unlabelled	1

Detux was unable to detect any indication of compromise for all the twenty-two (22) malware samples built for the x86-64 architecture. Detux, as described in sub-section 3.4.2.4 has virtual machine images for analysis of malware samples built for the x86, x86-64, ARM, MIPS and MIPSEL platforms. After the initial automatic analysis, the x86-64 architecture was specifically selected for the analysis of the 64-bit malware samples to eliminate the possibility of a failure in the built-in architecture detection, however Detux was not unable to detect any indication of compromise in the 64-bit samples.

Sub Question 3.5.4c	*Is Detux able to detect network, memory and operating system operations of malware samples after execution?*
Detux was able to detect indications of compromise on 106 out of 297 malware samples, However, it was unable to detect malicious activity on the malware samples built for x86-64 architecture.	

5.4.5 HaboMalHunter

There were three major categories during the dynamic analysis on HaboMalHunter. The first category was the malware samples invoking processes connecting to a TCP port locally on the system. The most popular of the TCP ports used was port 48101. This is a characteristic of a variant of the Mirai botnet malware. The malware process binds to the port to listen to incoming connections from the command centre (MalwareMustDie, 2016). One hundred and seventeen (117) malware samples exhibited this characteristic. Ninety-five (95) of them were of the Mirai family. The Xorddos and Gafgyt families were the other major contributors with five (5) and four (4) malware samples respectively. Five (5) of the malware samples were unlabelled, one of which had 96% similarity with a sample in the Mirai family. Table 5.6 shows the distribution of malware samples making this local process connections by malware family. The Dofloo, Turla, SSHgp, Grip, Miner, Scalper, Snoopy and PNScan families recorded one member each in this category.

Table 5. 6 Malware samples connecting to local TCP process by family (HaboMalHunter)

Family	Number of Samples
Mirai	95
Xorddos	5
Gafgyt	4
Dofloo	1
Turla	1
PnScan	1
SSHgo	1
Scalper	1
Miner	1
Grip	1
Snoopy	1
Unlabelled	5

The second major category is that of malware samples attempting to make connections to a variety of hosts. There are one hundred and twenty-one of these malware samples. The distribution per malware family class is shown in table 5.7. The major family categories with this indication of compromise were Mirai, Gafgyt and Xorddos with eight-nine (89), six (6) and three (3) malware samples respectively. There were fourteen (14) unlabelled malware samples. Four of the unlabelled malware samples were found to have some degree of similarity with other samples in the Mirai family. Two of them had 88% similarity while the remaining two had 90% and 94% similarity.

Table 5. 7 Malware samples connecting to multiple hosts by family (HaboMalHunter)

Family	Number of Samples
Mirai	89
Gafgyt	6
Xorddos	3
Znaich	1
Tsunami	1
Erebus	1
SSHgo	1
Mumblehard	1
Setag	1
Local	1
Adore	1
Nestea	1
Unlabelled	14

The last major category recorded six malware samples attempting to connect to command and control server IP addresses. The distribution is illustrated in table 5.8. Two of the samples are from the Mirai family while the Gafgyt and Binom families contributed one each. Two of the malware samples were unlabelled with one recorded as having 94% similarity with a malware sample in the Mirai family.

Table 5. 8 Malware samples connecting to control centres (HaboMalHunter)

Family	Number of Samples
Mirai	2
Gafgyt	1
Binom	1
Unlabelled	2

HaboMalhunter detected indications of compromise for sixteen (16) out of the twenty-two (22) malware samples built for the X86-64 architecture. It also detected malicious activity in two hundred and twenty-eight (228) out of the two hundred and seventy-five (275) malware samples targeted at the x86 platform.

Sub Question 3.5.5c	*Is HalMalHunter able to detect network, memory and operating system operations of malware samples after execution?*
Halmalhunter was able to detect indicators of compromise in 244 out of the 297 malware samples across the two CPU architectures.	

5.5 Answers to research hypotheses

There are twenty-seven (27) malware samples that all the sandboxes collectively could not detect any indications of compromise for. As a result, the first hypothesis; **the malware analysis sandboxes will collectively be able to detect indications of compromise from execution of all the malware samples** is rejected.

The second hypothesis that **all the analysis systems will have consistent analysis results for the malware samples executed and analysed** is accepted as the analysis results were consistent across the platforms for the samples that were identified as malicious by all samples. The classification of families by indicators of compromise was consistent across the platforms.

5.6 Conclusion

This chapter discussed results of the malware analysis by the sandbox platforms with respect to the malware family and CPU family classifications. The static and dynamic analysis results were discussed in detail and the remaining research questions as well as the research hypotheses were answered. Chapter 6 gives summary of the thesis as well as the contribution, limitations and possible future work that can follow on from the thesis.

6. Conclusions

6.1 Introduction

The popularity of the Microsoft Windows family of operating systems have made them the most researched platforms for malware analysis. Most malware samples in existence are targeted towards these systems. The server infrastructure space is dominated by Linux based systems. The processing power and relative stability of servers have made them lucrative targets for attackers. Servers also hold high value data. The increasing popularity IoT devices has also made Linux systems objects of attention to intruders. This chapter concludes the discussion on open source malware analysis sandboxes for Linux ELF binaries. The next section is a brief review of the previous chapters culminating in the conclusions reached. The contributions of this research are highlighted in the third section. The fourth section is a discussion of some of the limitations of the research that might have impacted the conclusions arrived at. The fifth section explores the possible research activities that can be embarked upon as a follow up to this research. The conclusions are restated in the final section.

6.2 Thesis Review

The research started with an introduction that discussed the motivation and aims of the research as well as the organisation of the thesis. A review of body of work that laid the foundation for this research was undertaken in the second chapter with a deep dive into the internals of the Linux operating system and the underlying concepts. The system organisation, system calls and libraries and forensic artefacts were discussed. The chapter also explored the topic of malware and malware analysis. The malware analysis types and their relative strengths and weaknesses were discussed. The foregoing topics on the Linux operating system and malware analysis drew upon key literature in those fields. Related literature to the topic of malware analysis sandboxes were reviewed. The review discussed the components of an analysis system and the importance of automation and instrumentation as well as the anti-analysis techniques of malware authors. Another aspect of this review touched on the existing research on the analysis of Linux ELF binaries. The existing research predominantly involved the use of data providers to generate properties of an ELF binary and the use of machine learning algorithms to predict the possibility of malicious intent. There was a dearth of research information on some of the malware analysis sandboxes that supported Linux ELF binary file analysis. The information available on these tools were author documentations and demonstrations at security events and trainings.

With the recognition of the need for malware samples, the third chapter further reviewed the current body of knowledge with respect the methods used to source malware samples and the instrumentation and analysis tools used. This review and that of second chapter exposed the gaps that exist in analysis of Linux malware samples with respect to analysis sandboxes. Adopting the practices already used in similar research, the processes for the sourcing malware samples were described and the steps for testing the malware samples on the sandboxes were also discussed. The research hypotheses and questions were stated in this chapter.

The results of the tests were discussed in the fourth chapter. The static and dynamic analysis results as well as reporting and instrumentation possibilities of the analysis sandboxes were presented. The research sub-questions on the reporting and automation functionality were answered in this chapter.

The fifth chapter discussed the results of the testing within the context of the characteristic of the dataset used. This discussion considered the CPU architectures the malware samples were built for and the malware family membership of the samples. The remaining research questions on the dynamic and static analysis features of the analysis sandboxes and the research hypothesis were answered in this chapter. While the analysis sandboxes had varying degrees of success in detecting indications of compromise; when multiple sandboxes have detected malicious intent during analysis of a malware sample, the indicators were consistent. Collectively, the sandboxes were unable to detect harmful properties in all the malware samples tested.

6.3 Contribution

This research explored the topic of malware analysis on the Linux operating system. This required an in-depth study of the system architecture of the operating system involving a discussion of the logical units. Malware are computer programs that are written with the objective to cause harm to their operating environment. Examination of these programs is a key step in understanding their methods of operation, providing answers to questions such as the specific vulnerabilities of the systems being exploited as well as the nature and the extent of the harm caused. These learnings form input into post infection activities. Some of these involve incident reports and disclosure to stake holders, improvement of exploited systems through patches or architectural overhaul, development of systems or signatures to prevent repeat occurrences. Malware analysis is challenged by the increasing rate of malware creation. This is aided by the ease of creating variants of existing malware samples in ways that change the signature hash but keep most of the functionality. Existing studies have proposed different ways to rapidly identify these variations using Context Triggered Piecewise Hashing and the importance of having an automated workflow to cope with the deluge of malware samples has also been addressed. Another challenge faced by malware analysis is the consistency of naming and classification of malware samples. Different anti-malware vendors refer to the same (by cryptographic hash) malware samples by different names. A method of labelling was proposed in one of the reviewed studies (Sebastián et al., 2016) that employed the aggregation of names used by anti-malware engines to specify a family label for malware samples.

The study of malware can be done without executing the sample. The process is known as static analysis and it involves an examination of the program structure. With reference to executable programs on the Linux operating system, an understanding of the ELF program structure is necessary. This file format as described in chapter 2 is the structure of executable binary files, core dump, shared libraries and process memory image. The ELF file describes how a program was compiled, the external libraries required and how it transformed into a process image in memory. The presence or absence of a requested linker library, the functions and symbol references can be used to determine the nature of a Linux binary file. These properties have been combined with machine leaning algorithms to predict malicious intent in malware samples as referenced in the related work section on chapter 2. The use of packers and the types used can also be inferred from static analysis.

A file can meet the requirements of the ELF specification and still sufficiently conceal information about its operation, making it impossible for static analysis to derive any meaningful observations. The removal of debugging information during compilation, compression and encryption of functions are some of the other anti-analysis methods that can be used to frustrate static analysis. The ability to execute these malware samples in a secure environment while observing them can be vital because

of the aforementioned challenges to static analysis. Chapter 2 discusses the file system, memory and process sub-system of the Linux operating system in depth. The study of a process through its interaction with memory, its system calls and signals to other processes, files and network entities are very important in dynamic analysis. This research also explored the forensic artefacts of the Linux system which serve as the staging points for the indicators of compromise. The different standalone tools for examination of malware samples during execution were investigated with focus of the staging environments in the form of hardware emulators and virtual machines. Some of the developments in lightweight emulators that avoid detection by malware such as Introlib and TTAnalyze were also discussed. Current research in Linux malware is dominated by systems tasked with prediction of malicious intent primarily through machine learning libraries and the classification of static, dynamic or hybrid qualities of the malware sample under examination. The importance of detection and prevention cannot be overstated; however, malware analysis through secure observation in a sandbox environment is invaluable in the inevitable situations of breach and compromise. The available literature on integrated dynamic analysis and sandbox environments for Linux malware samples were restricted to defunct projects such as Malwr and Anubis. Apart from Cuckoo sandbox, information about the other sandbox environments were in the domain of security blogs and conference demonstrations and trainings. Most of the references to Cuckoo were with respect to analysis of Windows based malware samples.

The relative effectiveness of these tools, with their pros and cons have not been addressed in the available literature. This research sought to bridge that gap by undertaking an empirical assessment of these tools against a field of malware samples from malware repositories and a honeypot setup. This assessment was done with known malware samples and the consistency of the results derived from these tests were compared with attributes already known about the malware samples. The results showed that the tools were generally consistent in the indications they detected even though the relative effectiveness varied. While the tools collectively were unable to detect all malicious attributes in all the malware samples, the exercise brought to light some improvements that can be added to the sandboxes. The answers to the research sub-questions can also serve as a guideline for malware researchers on the Linux platform on tool selection and the relative attributes of the tools available. The availability of the tools under various open source licenses and hosting on Github creates an avenue for adoption and improvement by other contributors.

HaboMalHunter was the most effective in detecting indicators of compromise, however, its operation is not automated. This limitation can potentially restrict its adoption because it would be unsuitable for high volume analysis. An analyst can choose to use it for one-off analysis based on its strengths in detection and reporting. A possible improvement to HaboMalHunter is the creation of a wrapper script to orchestrate the virtual machine during analysis so that a known good state can be reverted to after analysis. All the sandboxes could benefit from the multi architecture support offered by Detux through its use of the Qemu emulator. This will enhance the utility of these tools as Linux runs on a variety of architectures and a specific architecture might be the exclusive target execution environment for a malware sample.

6.4 Limitations

This section is an overview on some of the items that have possibly affected the conclusions of the research. This is an exploration of factors that might have hampered the testing and ways this could have been improved upon.

6.4.1 Diversity of dataset

An examination of the malware samples revealed that most of the samples were of the Mirai family or its variants. This would likely have contributed to the context fuzzy hash investigations that indicated that most of the pairs that were similar belonged to the same family. A more diverse dataset with respect to malware family classification might have challenged the outcome of that correlation.

The situation where most of the malware samples exhibited characteristics that were predominantly network related also limited the capabilities of the analysis sandboxes that could be tested. The limitation of the malware samples to just those built for the x86 and x86-64 platforms might have also limited the range of malware samples characteristics that could be tested. The popularity of IoT devices might have thrown up the possibility of a more diverse malware collection if other architectures especially those popular with embedded devices like ARM, MIPS, MIPSEL were considered. There were only twenty-two (22) malware samples built for the x86-64 platform so a conclusion on the ability of Detux to analyse malware samples built for the platform could not be arrived at.

The lack of variety in the sample pool also stems from the deployment of the honeypot as a single host. Having multiple hosts with different CPU architectures and functions would have attracted a more diverse pool of malware samples. The VirusTotal paid subscription service gives access to versatile query APIs for selection of malware samples by family, architecture and file type as well as date. This flexibility would allow for a variety of options in sample selection.

6.4.2 System libraries and hardware extensions

Some of the malware samples failed to execute due to missing libraries. Cross compilation of dynamic linker and loaders for the analysis environment would have allowed analysis of some of those malware samples. Some ransomware samples failed to run also due to absence of AES-NI CPU extension. Having a variety of CPU hardware architectures in the analysis environment will enable more success in malware execution and analysis.

6.4.3 Internet Access

There was a restriction of outgoing Internet access in the analysis and honeypot environments. In the former environment, the full capabilities of the malware samples might not be on full display with a lack of outgoing Internet access despite the role Inetsim played in emulating network and Internet services. The constraint on outgoing Internet access was to prevent the enlistment of the analysed virtual systems as bots used to cause harm to other systems. The lack of outgoing Internet access from the honeypot might have affected the value of the post infection analysis as well as the effectiveness of the entrapment.

6.5 Future Work

There are two areas of future research that can arise from this study. A focus on malware built for IoT devices will be an important study. The effectiveness of sandboxes in detecting indications of compromise in these devices would assist malware researchers in the selection of analysis tools. IoTPot and IoTBox are platforms that can be evaluated in this research (Pa et al., 2015).

While the use of the *string* utility and *Yara rules* helped determine some of the packers (12) used in the samples, there is still a gap on definitively telling when an ELF file is packed from static analysis. This is an area where further research would be of benefit to the security community. The creation of more Yara rules will also enhance this activity.

6.6 Conclusion

This chapter concludes the thesis with an overview of the discussion, the contributions and limitations of the thesis and research opportunities in the field of Linux malware analysis. The thesis was focussed on experimental investigation of the performance of sandboxes for Linux binary files. This is of importance because of the increasing popularity of Linux in the embedded hardware space as well as the its large installed base of servers. Five malware testing sandboxes - REMnux, Limon, Cuckoo, Detux and HaboMalHunter - were tested for their ability to detect indicators of compromise in malware samples. The presence of obfuscation (packing) techniques was also tested. Security research firms have found that most malware samples are variations of existing ones. The ability of the sandboxes to make this association amongst the malware samples was also evaluated. The reporting and automation features available to the sandboxes were also examined. The importance of automation is due to the rate at which malware samples are being created which can make manual analysis unfeasible.

Out of two hundred and ninety-seven (297) malware samples, the sandboxes collectively did not detect indicators of compromise in twenty-seven (27) of the samples. Limon was found to be the only sandbox capable of detecting metamorphic variants through its use of fuzzy hashing techniques. The sandboxes were found to be consistent in the indicators found where indicators of malice were discovered, with HaboMalhunter recording the most detections. Limon, Detux and Cuckoo had workflows that could be automated because they automatically handled the creation and the restoration to the default state of the virtual environments for each sample analysis.

7. References

504ensicslabs. (2017). Linux Memory Extractor. Retrieved from https://github.com/504ensicsLabs/LiME

AhnLab. (2014). All You Need to Know about the Latest Linux Malware. Retrieved from http://global.ahnlab.com/global/upload/download/documents/1501086836841003.pdf

Alam, S., Horspool, R. N., Traore, I., & Sogukpinar, I. (2015). A framework for metamorphic malware analysis and real-time detection. *Computers & Security*, *48*, 212-233. doi:10.1016/j.cose.2014.10.011

Aljaedi, A., Lindskog, D., Zavarsky, P., Ruhl, R., & Almari, F. (2011). *Comparative analysis of volatile memory forensics: live response vs. memory imaging.* Paper presented at the 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third International Conference on Social Computing (SocialCom), Boston, MA, USA.

Anubis. (2015). We are discontinuing the Anubis and Wepawet services. Retrieved from http://anubis.iseclab.org/

Aslan, Ö., & Samet, R. (2017). Investigation of Possibilities to Detect Malware Using Existing Tools. *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, Tunisia, 2018, pp. 1277-1284. doi:10.1109/AICCSA.2017.24

Asmitha, K., & Vinod, P. (2014). A machine learning approach for linux malware detection. *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Ghaziabad, 2014, pp. 825-830. doi: 10.1109/ICICICT.2014.6781387

Asmitha, K. A., & Vinod, P. (2014). Linux malware detection using non-parametric statistical methods. *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, New Delhi, 2014, pp. 356-361. doi: 10.1109/ICACCI.2014.6968611

Azab, A., Layton, R., Alazab, M., & Oliver, J. (2014). Mining Malware to Detect Variants. *2014 Fifth Cybercrime and Trustworthy Computing Conference*, Auckland, 2014, pp. 44-53. doi: 10.1109/CTC.2014.11

Bakhshayeshi, R., Akbari, M. K., & Javan, M. S. (2014). Performance analysis of virtualized environments using HPC Challenge benchmark suite and Analytic Hierarchy Process. *2014 Iranian Conference on Intelligent Systems (ICIS)*, Bam, 2014, pp. 1-6. doi: 10.1109/IranianCIS.2014.6802585

Bayer, U., Kirda, E., & Kruegel, C. (2010). Improving the efficiency of dynamic malware analysis. *Proceedings of the 2010 ACM Symposium on Applied Computing* (SAC '10).

ACM, New York, NY, USA, 1871-1878. DOI: https://doi.org/10.1145/1774088.1774484

Bayer, U., Kruegel, C., & Kirda, E. (2006). *TTAnalyze: A tool for analyzing malware.* Paper presented at the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference, Hamburg.

Bist, A. S. (2014). Detection of metamorphic viruses: A survey. *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, New Delhi, 2014, pp. 1559-1565. doi: 10.1109/ICACCI.2014.6968246

Botacin, M. F., de Geus, P. L., & Grégio, A. R. A. (2017). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 1-12. doi: 10.1007/s11416-017-0292-8

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. doi:10.1007/s11416-015-0247-x

Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel: from I/O ports to process management*: Sebastopol, CA: O'Reilly Media, Inc.

Boyle, R. J., & Panko, R. R. (2014). *Corporate computer security*: Upper Saddle River, NJ: Prentice Hall.

Bryant, R. (2016). *Policing digital crime*: New York, NY: Routledge.

Bryant, R. E., & O'Hallaron, D. R. (2015). Computer Systems: A Programmer's Perspective. Upper Saddle River, NJ: Pearson Education Inc.

Canonical Ltd. (2017). Ubuntu release end of life. Retrieved from https://www.ubuntu.com/info/release-end-of-life

Case, A., & Richard, G. G. (2017). Memory forensics: The path forward. *Digital Investigation, 20*, 23-33. doi:https://doi.org/10.1016/j.diin.2016.12.004

Choudhary, S. P., & Vidyarthi, M. D. (2015). A Simple Method for Detection of Metamorphic Malware using Dynamic Analysis and Text Mining. *Procedia Computer Science, 54*, 265-270. doi:https://doi.org/10.1016/j.procs.2015.06.031

Cohen, M. (2016). The pmem suite of memory acquisition tools. Retrieved from http://blog.rekall-forensic.com/2016/05/the-pmem-suite-of-memory-acquisition.html

Cohen, M., Garfinkel, S., & Schatz, B. (2009). Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. *Digital Investigation, 6*, S57-S68. https://doi.org/10.1016/j.diin.2009.06.010

Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux Device Drivers: Where the Kernel Meets the Hardware*: Sebastopol, CA: O'Reilly Media, Inc.

Cozzi, E., Graziano, M., Fratantonio, Y., & Balzarotti, D. (2018). Understanding Linux Malware. *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, 2018, pp. 161-175. doi: 10.1109/SP.2018.00054

Cuckoo Foundation. (2015). Cuckoo Sandbox Book. Retrieved from http://docs.cuckoosandbox.org/en/latest/

Daly, M. K. (2009). Advanced persistent threat. *Usenix, November 4, 2009*.

Damri, G., & Vidyarthi, D. (2016). *Automatic dynamic malware analysis techniques for Linux environment.* Paper presented at the 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India.

De Andrade, C. A. B., De Mello, C. G., & Duarte, J. C. (2013). Malware automatic analysis. *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, Ipojuca, 2013, pp. 681-686. doi: 10.1109/BRICS-CCI-CBIC.2013.119

Deng, Z., Xu, D., Zhang, X., & Jiang, X. (2012). Introlib: Efficient and transparent library call introspection for malware forensics. *Digital Investigation, 9*, S13-S23. https://doi.org/10.1016/j.diin.2012.05.013

Detux Sandbox. (2018). The Multiplatform Linux Sandbox. Retrieved from https://github.com/detuxsandbox/detux

Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). *Ether: malware analysis via hardware virtualization extensions*. Paper presented at the Proceedings of the 15th ACM conference on Computer and communications security, Alexandria, Virginia, USA. https://doi.org/10.1145/1455770.1455779

DistroWatch. (2017). Top Ten Distributions: An overview of today's top distributions. Retrieved from https://distrowatch.com/dwres.php?resource=major

Dr Web. (2018). Linux.BtcMine.26. Retrieved from https://vms.drweb.com/virus/?i=15743486&lng=en

Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., . . . Payer, M. (2014). *The matter of heartbleed.* Paper presented at the Proceedings of the 2014 Conference on Internet Measurement Conference, Vancouver, BC, Canada, pp 475-488. https://doi.org/10.1145/2663716.2663755

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys, 44*(2), 6-6:42. doi:10.1145/2089125.2089126

Ehrenfeld, J. M. (2017). Wannacry, cybersecurity and health information technology: A time to act. *Journal of medical systems, 41*(7), 104. https://doi.org/10.1007/s10916-017-0752-1

F-Secure. (2018). Scalper. Retrieved from https://www.f-secure.com/v-descs/scalper.shtml

Ferrie, P. (2016). Attacks on virtual machine emulators. *Symantec Advanced Threat Research*.

Fortiguard. (2018a). Linux/Ddostf!tr. Retrieved from https://fortiguard.com/encyclopedia/virus/7433648

Fortiguard. (2018b). W32/Multi.MIBSUN!tr.bdr. Retrieved from https://fortiguard.com/encyclopedia/virus/7434810

Free Software Foundation Inc. (2017). GDB: The GNU Project Debugger. Retrieved from https://www.gnu.org/software/gdb/

freedesktop.org. (2017). Systemd System and Service Manager. Retrieved from https://www.freedesktop.org/wiki/Software/systemd/

Gandotra, E., Bansal, D., & Sofat, S. (2016). Zero-day malware detection. *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, Patna, 2016, pp. 171-175.
doi: 10.1109/ISED.2016.7977076

GFI Software. (2017). GFI Software. Retrieved from https://www.gfi.com/products-and-solutions/all-products

Google. (2017). Virus Total. Retrieved from https://www.virustotal.com/en/about/

Gouda, M. G., & Liu, A. X. (2005, 28 June-1 July 2005). A model of stateful firewalls and its properties. *2005 International Conference on Dependable Systems and Networks (DSN'05)*, Yokohama, Japan, 2005, pp. 128-137.
doi: 10.1109/DSN.2005.9

Grinberg, M. (2014). *Flask web development: developing web applications with python*: Sebastopol, CA: O'Reilly Media, Inc.

Guarnizo, J., Tambe, A., Bunia, S. S., Ochoa, M., Tippenhauer, N., Shabtai, A., & Elovici, Y. (2017). SIPHON: Towards ScalableHigh-Interaction Physical Honeypots. *arXiv preprint arXiv:1701.02446*.

Han, K. S., Lim, J. H., Kang, B., & Im, E. G. (2015). Malware analysis using visualized images and entropy graphs. *International Journal of Information Security, 14*(1), 1-14. doi:10.1007/s10207-014-0242-0

Hex-Rays. (2016). About IDA. Retrieved from https://www.hex-rays.com/products/ida/

Hungenberg, T., & Eckert, M. (2017). INetSim: Internet Services Simulation Suite. Retrieved from http://www.inetsim.org/about.html

Insecure. (2018). Nestea "Off By One" attack. Retrieved from http://insecure.org/sploits/linux.PalmOS.nestea.html

Islam, R., Tian, R., Batten, L. M., & Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications, 36*(2), 646-656. doi:https://doi.org/10.1016/j.jnca.2012.10.004

Juniper Networks. (2017a). About Juniper. Retrieved from https://www.juniper.net/us/en/company/

Juniper Networks. (2017b). EX Series. Retrieved from https://www.juniper.net/us/en/products-services/switching/ex-series/compare?p=EX2200-C

Juniper Networks. (2017c). JTAC Recommended Junos Software Versions. Retrieved from https://kb.juniper.net/InfoCenter/index?page=content&id=KB21476&actp=METADATA

Juniper Networks. (2017d). SRX Series. Retrieved from https://www.juniper.net/us/en/products-services/security/srx-series/compare?p=SRX300

Kerrisk, M. (2010). *The Linux programming interface*: San Francisco, CA: No Starch Press.

Kirat, D., Vigna, G., & Kruegel, C. (2014). *BareCloud: Bare-metal Analysis-based Evasive Malware Detection.* Proceedings of the 23rd USENIX conference on Security Symposium, pp 287-301.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation, 3*, 91-97. https://doi.org/10.1016/j.diin.2006.06.015

Ligh, M., Adair, S., Hartstein, B., & Richard, M. (2010). *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. New York, NY: Wiley Publishing.

Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. New York, NY:John Wiley & Sons.

Linux Kernel Organization. (2016). Ext4 (and Ext2/Ext3) Wiki Retrieved from https://ext4.wiki.kernel.org/index.php/Main_Page

Linux Programmer's Manual. (2016). PTRACE(2). Retrieved from http://man7.org/linux/man-pages/man2/ptrace.2.html

Love, R. (2010). *Linux kernel development*. Upper Saddle River, NJ: Pearson Education.

Mairh, A., Barik, D., Verma, K., & Jena, D. (2011). *Honeypot in network security: a survey*. Paper presented at the Proceedings of the 2011 international conference on communication, computing & security. https://doi.org/10.1145/1947940.1948065

Malicia Lab. (2018). AVClass malware labeling tool. Retrieved from https://github.com/malicialab/avclass

Malin, C. H., Casey, E., & Aquilina, J. M. (2008). *Malware forensics: investigating and analyzing malicious code*. Burlington, MA: Syngress.

Malin, C. H., Casey, E., & Aquilina, J. M. (2013). *Malware forensics field guide for Linux systems: digital forensics field guides*. Boston, MA: Newnes.

MalwareMustDie. (2016). MMD-0056-2016 - Linux/Mirai, how an old ELF malcode is recycled.. . Retrieved from http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html

Malwr. (2016). Malwr. Retrieved from https://malwr.com/submission/

Marek, J. M. (2002). Plain English: Risks of Java Applets and Microsoft ActiveX Controls. *SANS Institute*, *March* 4 (2002), p 4.

McAfee. (2018). Virus Profile: Linux/Binom. Retrieved from https://home.mcafee.com/virusinfo/virusprofile.aspx?key=130506#none

Mehdi, S. B., Tanwani, A. K., & Farooq, M. (2009). *IMAD: in-execution malware analysis and detection*. Paper presented at the Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, Quebec, Canada. https://doi.org/10.1016/j.ins.2011.09.016

Mehra, M., & Pandey, D. (2016). Event triggered malware: A new challenge to sandboxing. *2015 Annual IEEE India Conference (INDICON)*, New Delhi, 2015, pp. 1-6. doi: 10.1109/INDICON.2015.7443327

Messier, R. (2015). *Operating System Forensics*. Burlington, MA: Syngress.

Microsoft. (2018a). Backdoor:Linux/Setag.C. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor%3ALinux%2FSetag.C

Microsoft. (2018b). Exploit:Linux/Bonk.E. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Exploit%252525253aLinux%252525252fBonk.E&ThreatID=-2147390635

Microsoft. (2018c). Exploit:Linux/Local.AO. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Exploit%3ALinux%2FLocal.AO

Microsoft. (2018d). Exploit:Linux/Race.H. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Exploit%3ALinux%2FRace.H&ThreatID=-2147390548

Microsoft. (2018e). Trojan:Linux/Midav.A. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan%25253aLinux%25252fMidav.A&ThreatID=-2147391089

Microsoft. (2018f). Virus:Linux/Grip!gen0. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus%3ALinux%2FGrip!gen0

Microsoft. (2018g). Virus:Linux/Thou.B. Retrieved from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus%3ALinux%2FThou.B&ThreatID=-2147364920

Miller, C., Glendowne, D., Cook, H., Thomas, D., Lanclos, C., & Pape, P. (2017). Insights gained from constructing a large scale dynamic analysis platform. *Digital Investigation, 22*, S48-S56. https://doi.org/10.1016/j.diin.2017.06.007

Mokube, I., & Adams, M. (2007). *Honeypots: concepts, approaches, and challenges.* ACM-SE 45 Proceedings of the 45th annual southeast regional conference, pp 321-326. https://doi.org/10.1145/1233341.1233399

Monnappa, K. (2015). Automating Linux Malware Analysis Using Limon Sandbox. *Black Hat Europe 2015*.

Monnappa K A. (2015). Setting up Limon Sandbox for Analyzing Linux Malwares. Retrieved from http://malware-unplugged.blogspot.co.nz/2015/11/setting-up-limon-sandbox-for-analyzing.html

Moser, A., Kruegel, C., & Kirda, E. (2007, 20-23 May 2007). Exploring Multiple Execution Paths for Malware Analysis. *2007 IEEE Symposium on Security and Privacy (SP '07)*, Berkeley, CA, 2007, pp. 231-245. doi: 10.1109/SP.2007.17

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). *Malware images: visualization and automatic classification*. Paper presented at the Proceedings of the 8th International Symposium on Visualization for Cyber Security, Pittsburgh, Pennsylvania, USA. https://doi.org/10.1145/2016904.2016908

Nataraj, L., Yegneswaran, V., Porras, P., & Zhang, J. (2011). *A comparative assessment of malware classification using binary texture analysis and dynamic analysis*. Paper presented at the Proceedings of the 4th ACM workshop on Security and artificial intelligence, Chicago, Illinois, USA. https://doi.org/10.1145/2046684.2046689

Nelson, B., Phillips, A., & Steuart, C. (2014). *Guide to computer forensics and investigations*. Boston, MA: Cengage Learning.

Netscraft Ltd. (2017). September 2017 Web Server Survey. Retrieved from

https://news.netcraft.com/archives/2017/09/11/september-2017-web-server-survey.html

Neugschwandtner, M., Comparetti, P. M., & Platzer, C. (2011). *Detecting malware's failover C&C strategies with squeeze.* Paper presented at the Proceedings of the 27th annual computer security applications conference, Orlando, FL, USA. https://doi.org/10.1145/2076732.2076736

Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., & Nguyen, H. D. (2009). MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. *2009 Annual Computer Security Applications Conference*, Honolulu, HI, 2009, pp. 441-450. doi: 10.1109/ACSAC.2009.48.

NJCCIC. (2018). PNScan. Retrieved from https://www.cyber.nj.gov/threat-profiles/trojan-variants/pnscan

Noor, M., Abbas, H., & Shahid, W. B. (2018). Countering cyber threats for industrial applications: An automated approach for malware evasion detection and analysis. *Journal of Network and Computer Applications, 103*, 249-261. doi:https://doi.org/10.1016/j.jnca.2017.10.004

Offensive Security. (2017). Offensive Security Training, Certifications and Services. Retrieved from https://www.offensive-security.com/

Oktavianto, D., & Muhardianto, I. (2013). *Cuckoo Malware Analysis*. Birmingham, UK: Packt Publishing Ltd.

Openshift. (2018). Kernel Virtual Machine. Retrieved from https://www.linux-kvm.org/page/Main_Page

Oracle. (2017). MySQL. Retrieved from https://www.mysql.com/

Ouchn, N. (2011). GFI Sandbox (formerly CWSandbox) The Malware Analysis Tool v3.2 released. Retrieved from http://www.toolswatch.org/2011/04/gfi-sandbox%E2%84%A2-formerly-cwsandbox-the-malware-analysis-tool-v3-2-released/

Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., & Rossow, C. (2015). IoTPOT: Analysing the rise of IoT compromises. WOOT'15 Proceedings of the 9th USENIX Conference on Offensive Technologies, pp 9-9.

Pektaş, A., & Acarman, T. (2017). Classification of malware families based on runtime behaviors. *Journal of Information Security and Applications, 37*, 91-100. doi:10.1016/j.jisa.2017.10.005

Provataki, A., & Katos, V. (2013a). Differential malware forensics. *Digital Investigation, 10*, 311-322. doi:10.1016/j.diin.2013.08.006

Provataki, A., & Katos, V. (2013b). Differential malware forensics. *Digital Investigation,*

10(4), 311-322. doi:https://doi.org/10.1016/j.diin.2013.08.006

QEMU. (2017). QEMU - The FAST! processor emulator. Retrieved from www.qemu.org

Radare. (2017). Radare. Retrieved from http://rada.re/r/index.html

Redhat. (2018). Erebus Malware. Retrieved from https://access.redhat.com/solutions/3094421

Rekall Forensics. (2017). Rekall Forensics. Retrieved from http://www.rekall-forensic.com/

REMnux Documentation Team. (2017). REMnux Documentation. Retrieved from https://remnux.org/docs/

Rieck, K., Holz, T., Willems, C., Düssel, P., & Laskov, P. (2008). *Learning and Classification of Malware Behavior*. Paper presented at the 5th International Conference of Detection of Intrusions and Malware, and Vulnerability Assessment, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-70542-0_6

Ronancher, A. (2017). Flask: Web development, one drop at a time. Retrieved from http://flask.pocoo.org/

Rosen, R. (2014). Internet Control Message Protocol (ICMP) *Linux Kernel Networking: Implementation and Theory* (pp. 37-61). Berkeley, CA: Apress.

Rubio Ayala, S. (2017). An automated behaviour-based malware analysis method based on free open source software. http://hdl.handle.net/10609/66365

SANS. (2017). SANS Institute. Retrieved from https://www.sans.org/

Sarantinos, N., Benzaïd, C., Arabiat, O., & Al-Nemrat, A. (2016). Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities. *2016 IEEE Trustcom/BigDataSE/ISPA*, Tianjin, 2016, pp. 1782-1787. doi: 10.1109/TrustCom.2016.0274

Sebastián, M., Rivera, R., Kotzias, P., & Caballero, J. (2016). *Avclass: A tool for massive malware labeling. International Symposium on Research in Attacks, Intrusions, and Defenses* (pp. 230-253). Springer, Cham.

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

Shahzad, F., & Farooq, M. (2012). ELF-Miner: using structural knowledge and data mining methods to detect new (Linux) malicious executables. *Knowledge and Information Systems*, *30*(3), 589-612. doi:10.1007/s10115-011-0393-5

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

Sharma, A., & Sahay, S. K. (2014). Evolution and detection of polymorphic and metamorphic malwares: A survey. *arXiv preprint arXiv:1406.7061*.

Shijo, P. V., & Salim, A. (2015). Integrated Static and Dynamic Analysis for Malware Detection. *Procedia Computer Science, 46*, 804-811. https://doi.org/10.1016/j.procs.2015.02.149

Shipp, R. (2018). Online Scanners and Sandboxes. Retrieved from https://github.com/rshipp/awesome-malware-analysis#online-scanners-and-sandboxes

Shotts Jr, W. E. (2012). *The linux command line: A complete introduction*. San Francisco, CA: No Starch Press.

Sikorski, M., & Honig, A. (2012). *Practical malware analysis: the hands-on guide to dissecting malicious software*. San Francisco, CA: No Starch Press.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2013). *Operating system concepts* (9th ed.). Hoboken, NJ: Wiley.

Singhal, A., & Ou, X. (2017). Security Risk Analysis of Enterprise Networks Using Probabilistic Attack Graphs *Network Security Metrics* (pp. 53-73). Cham: Springer International Publishing.

Skoudis, E., & Zeltser, L. (2004). *Malware: Fighting malicious code*. Upper Saddle River, NJ: Prentice Hall

Sochor, T., & Zuzcak, M. (2015). *Attractiveness study of honeypots and honeynets in internet threat detection.* Paper presented at the International Conference on Computer Networks. doi:10.1007/978-3-319-19419-6_7

Sourceforge. (2016). About AIDE. Retrieved from http://aide.sourceforge.net/

Sourceware. (2018a). Readelf. Retrieved from https://sourceware.org/binutils/docs/binutils/readelf.html

Sourceware. (2018b). SystemTap Overview. Retrieved from https://sourceware.org/systemtap/

Strace. (n.d.). strace(1) - Linux man page. Retrieved from https://linux.die.net/man/1/strace

Suiche, M. (2016). Your favorite Memory Toolkit is back… FOR FREE! Retrieved from https://blog.comae.io/your-favorite-memory-toolkit-is-back-f97072d33d5c

Symantec. (2017). *Internet Security Threat Report*. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf

Symantec. (2018a). *Internet Security Threat Report*. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf

Symantec. (2018b). Linux.Adore.Worm. Retrieved from https://www.symantec.com/security-center/writeup/2001-040516-0452-99

Symantec. (2018c). Linux.Dofloo. Retrieved from https://www.symantec.com/security-center/writeup/2015-070812-0012-99

Symantec. (2018d). Linux.Gafgyt. Retrieved from https://www.symantec.com/security-center/writeup/2014-100222-5658-99?om_rssid=sr-

Symantec. (2018e). Linux.Mirai. Retrieved from https://www.symantec.com/security-center/writeup/2016-112905-2551-99

Symantec. (2018f). Linux.Mumblehard. Retrieved from https://www.symantec.com/security-center/writeup/2015-050707-4128-99

Symantec. (2018g). Linux.Turla. Retrieved from https://www.symantec.com/security-center/writeup/2014-121014-2918-99

Symantec. (2018h). Linux.Xorddos. Retrieved from https://www.symantec.com/security-center/writeup/2015-010823-3741-99

Sysdig. (2017). Sysdig. Retrieved from https://www.sysdig.org/

Szor, P. (2005). *The art of computer virus research and defense*. Upper Saddle River, NJ: Pearson Education .

Talos. (2018). Forgot About Default Accounts? No Worries, GoScanSSH Didn't Retrieved from https://blog.talosintelligence.com/2018/03/goscanssh-analysis.html

Tanenbaum, A. S., & Bos, H. (2014). *Modern operating systems*. Upper Saddle River, NJ: Prentice Hall

Tencent. (2018). HaboMalHunter: Habo Linux Malware Analysis System. Retrieved from https://github.com/Tencent/HaboMalHunter

The Apache Software Foundation. (2017). The Apache HTTP Server Project. Retrieved from https://httpd.apache.org/ABOUT_APACHE.html

The Volatility Foundation. (2014). VOLATILITY FOUNDATION. Retrieved from http://www.volatilityfoundation.org/

Tian, R., Islam, R., Batten, L., & Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. *2010 5th International Conference on Malicious and Unwanted Software*, Nancy, Lorraine, 2010, pp. 23-30. doi: 10.1109/MALWARE.2010.5665796

Tirli, H., Pektas, A., Falcone, Y., & Erdogan, N. (2013). *Virmon: a virtualization-based automated dynamic malware analysis system.* Paper presented at the 6th International Information Security & Cryptology Conference, Istanbul, Turkey, pp. 1-6.

Tool Interface Standards Committee. (2001). Executable and Linkable Format (ELF). *Specification, Unix System Laboratories, 1*(1), 1-20.

Trend Micro. (2018a). ELF_SVAT.A. Retrieved from https://www.trendmicro.com/vinfo/us/threat-encyclopedia/archive/malware/elf_svat.a

Trend Micro. (2018b). UNIX_FLOODDOS.A. Retrieved from https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/unix_flooddos.a

Trend Micro. (2018c). UNIX_SNOOPY.C. Retrieved from http://www.trendmicro.com.hk/vinfo/hk/threat-encyclopedia/archive/malware/unix_snoopy.c

Tsyganok, K., Tumoyan, E., Babenko, L., & Anikeev, M. (2012). *Classification of polymorphic and metamorphic malware samples based on their behavior*. Paper presented at the Proceedings of the Fifth International Conference on Security of Information and Networks, Jaipur, India, pp. 111-116

UBM Tech. (2015). Blackhat Europe 2015. Retrieved from https://www.blackhat.com/eu-15/briefings.html#automating-linux-malware-analysis-using-limon-sandbox

Ullrich, J. B. (2016). The Short Life of a Vulnerable DVR Connected to the Internet. Retrieved from https://isc.sans.edu/forums/diary/The+Short+Life+of+a+Vulnerable+DVR+Connected+to+the+Internet/21543/

Vasilescu, M., Gheorghe, L., & Tapus, N. (2014, 11-13 Sept. 2014). Practical malware analysis based on sandboxing. *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, Chisinau, 2014, pp. 1-6.
doi: 10.1109/RoEduNet-RENAM.2014.6955304

Virus Share. (2017). Because sharing is caring. Retrieved from https://virusshare.com/

VirusTotal. (2017). Malware analysis sandbox aggregation: Welcome Tencent HABO! . Retrieved from http://blog.virustotal.com/2017/11/malware-analysis-sandbox-aggregation.html

VX Heaven. (2017). VX Heaven. Retrieved from http://vxheaven.org/

Wade, S. M. (2011). SCADA Honeynets: The attractiveness of honeypots as critical infrastructure security tools for the detection and analysis of advanced threats. Graduate Theses and Dissertations. 12138. https://lib.dr.iastate.edu/etd/12138

Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D. A., . . . Viola, I. (2015). *A*

survey of visualization systems for malware analysis. Paper presented at the Eurographics Conference on Visualization (EuroVis) (2015), At Cagliari, Italy.

Ward, B. (2014). *How Linux works: What every superuser should know*. San Francisco, CA: No Starch Press

Willems, C., Holz, T., & Freiling, F. (2007). Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy, 5*(2), 32-39. doi:10.1109/MSP.2007.45

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., . . . Rossow, C. (2016). *SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion.* Paper presented at the Research in Attacks, Intrusions, and Defenses. RAID 2016. Lecture Notes in Computer Science, vol 9854. https://doi.org/10.1007/978-3-319-45719-2_8

You, I., & Yim, K. (2010, 4-6 Nov. 2010). *Malware Obfuscation Techniques: A Brief Survey. 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, Fukuoka, 2010, pp. 297-300. doi: 10.1109/BWCCA.2010.85 .

Younge, A. J., Henschel, R., Brown, J. T., Laszewski, G. v., Qiu, J., & Fox, G. C. (2011, 4-9 July 2011). Analysis of Virtualization Technologies for High Performance Computing Environments. 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, 2011, pp. 9-16. doi: 10.1109/CLOUD.2011.29

Zeltser, L. (2017a). Free Automated Malware Analysis Sandboxes and Services. Retrieved from https://zeltser.com/automated-malware-analysis/

Zeltser, L. (2017b). Free Toolkits for Automating Malware Analysis. Retrieved from https://zeltser.com/malware-analysis-tool-frameworks/

Zeltser, L. (2017c). Malware Sample Sources for Researchers. Retrieved from https://zeltser.com/malware-sample-sources/

Zoltan, B. (2016). Malware Analysis Sandbox Testing Methodology. *The Journal On Cybercrime & Digital Investigations, 1*(1). doi:10.18464/cybin.v1i1.3

APPENDIX

List of malware samples

MD5	Date of first submission	Scan Summary
374d0e146452a390bea075f1f6530cde	2016-10-03 18:32:12 UTC	33 of 59 positively flagged this sample as malicious
27d857e12b9be5d43f935b8cc86eaabf	2017-06-13 01:19:11 UTC	39 of 57 positively flagged this sample as malicious
320adee47e53823a1be8a335e4beb246	2015-07-24 08:40:50 UTC	36 of 59 positively flagged this sample as malicious
a44aa4f46a9dc68c97142f3825431c29	2017-10-19 20:51:32 UTC	30 of 59 positively flagged this sample as malicious
22dc1db1a876721727cca37c21d31655	2015-11-07 05:48:50 UTC	40 of 58 positively flagged this sample as malicious
483b322b42835227d98f523f9df5c6fc	2016-11-27 11:26:26 UTC	31 of 59 positively flagged this sample as malicious
61e0618a3984cbcb75b329beb069e0e9	2017-10-07 05:32:55 UTC	37 of 60 positively flagged this sample as malicious
0c1aa91e8cae4352eb16d93f17c0da2b	2017-04-22 11:00:31 UTC	36 of 60 positively flagged this sample as malicious
a7a1abb5d8e87fc670b6841a805103df	2017-04-22 11:00:37 UTC	37 of 59 positively flagged this sample as malicious
19fbd8cbfb12482e8020a887d6427315	2014-12-06 13:35:37 UTC	35 of 59 positively flagged this sample as malicious
132ba54b1b187a38a455dd27c1e74d62	2015-01-07 14:43:58 UTC	41 of 60 positively flagged this sample as malicious
3437bd29e5c8fe493603581dbb0285c7	2014-06-06 20:05:37 UTC	36 of 60 positively flagged this sample as malicious
009a9c6eee3e0fa532cfebe6a52be113	2017-10-20 11:07:41 UTC	38 of 60 positively flagged this sample as malicious
1975ff1586f0115e89fa1fe72708939a	2014-12-18 01:37:39 UTC	41 of 60 positively flagged this sample as malicious
1cadf5fc7f0729bb660aeb60a9e2207f	2015-08-05 14:24:19 UTC	39 of 60 positively flagged this sample as malicious
12770c550d06e95e0d580fc7dc287647	2016-02-02 10:20:11 UTC	31 of 58 positively flagged this sample as malicious
0b7630ead879da12b74b2ed7566da2fe	2014-12-16 19:31:24 UTC	38 of 59 positively flagged this sample as malicious
1e19b857a5f5a9680555fa9623a88e99	2015-11-07 05:48:51 UTC	43 of 60 positively flagged this sample as malicious
005c22845f8b7d92702ee3a5c37489cf	2017-10-10 12:12:24 UTC	22 of 60 positively flagged this sample as malicious
01adaa2fc9412ee02cb7adde58cd4fe1	2017-09-14 05:44:49 UTC	21 of 58 positively flagged this sample as malicious
024c094fff1f93ff68512d86b07d4f33	2017-08-17 01:02:57 UTC	36 of 60 positively flagged this sample as malicious
03a1e6c72c9978158a954c85556f74d1	2017-09-14 07:07:14 UTC	21 of 58 positively flagged this sample as malicious
0403fc76b30b735ae5881a06abe539a6	2016-06-24 18:32:54 UTC	37 of 58 positively flagged this sample as malicious
04e634a2aade2b99473c26be78c1bfd6	2017-09-10 01:34:37 UTC	26 of 58 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
056e5e432e4a57a44bb96498b649f1ee	2017-09-20 00:23:39 UTC	22 of 59 positively flagged this sample as malicious
071a632b2c9babfea998df852e0dc1f0	2017-06-19 19:26:31 UTC	13 of 56 positively flagged this sample as malicious
07b1295477c295540f08ecf07bbcfe30	2017-09-14 05:44:39 UTC	7 of 58 positively flagged this sample as malicious
086fef17ec295a3f8d469f3893246f3c	2017-06-20 00:01:43 UTC	11 of 57 positively flagged this sample as malicious
09f0b3699f57217ab669c47962a7aa4f	2017-09-25 07:26:10 UTC	28 of 59 positively flagged this sample as malicious
0c1aa91e8cae4352eb16d93f17c0da2b	2017-04-22 11:00:31 UTC	36 of 60 positively flagged this sample as malicious
0cdc43091b4b10ff0b4d574c841b803a	2017-09-14 05:06:08 UTC	21 of 58 positively flagged this sample as malicious
0dbe035cb9c5901dcacfe6505fdfb7e5	2017-09-14 04:56:30 UTC	9 of 58 positively flagged this sample as malicious
0de12e358555c92da2bf8dca21e6f54b	2017-08-18 22:56:19 UTC	26 of 57 positively flagged this sample as malicious
0deb84fce9da7a3561994af4d8ee8a83	2017-09-14 05:59:11 UTC	20 of 57 positively flagged this sample as malicious
0e104b109f86d7e5005e4ea7f3d27722	2017-09-14 06:49:13 UTC	9 of 57 positively flagged this sample as malicious
0f4825035617c6b08c6a9a4b0def31bc	2017-10-10 19:01:56 UTC	22 of 60 positively flagged this sample as malicious
0f60b0b617b04f1698526ac102787592	2017-08-21 09:45:08 UTC	20 of 57 positively flagged this sample as malicious
0f795e0079bf208b82470e09a7675f83	2017-09-09 04:46:30 UTC	27 of 58 positively flagged this sample as malicious
10c0e8ad9f935d33f396d99d0ba667a6	2016-06-18 18:32:34 UTC	26 of 60 positively flagged this sample as malicious
116ebab5d8eaa36862963b92cc80d384	2017-08-18 23:51:31 UTC	28 of 58 positively flagged this sample as malicious
127eacc6f5306caa43a600e428e9002f	2017-09-21 01:37:03 UTC	9 of 58 positively flagged this sample as malicious
12847fb913333b5bfe9e3d48657d78ec	2017-08-15 17:11:32 UTC	26 of 58 positively flagged this sample as malicious
12faad000218496fae305c88a3381494	2017-06-20 00:01:39 UTC	8 of 57 positively flagged this sample as malicious
137c1520b37dfc3ce5072be7995c96fc	2016-06-24 18:32:51 UTC	34 of 58 positively flagged this sample as malicious
13a704a8c4d463523e7a8b49527f4178	2017-10-11 22:18:17 UTC	26 of 60 positively flagged this sample as malicious
14775d0fb2fe528d59046278077ba845	2017-09-14 05:12:06 UTC	21 of 58 positively flagged this sample as malicious
157679ac46d453489aba544e266ae5af	2017-09-14 05:47:07 UTC	9 of 57 positively flagged this sample as malicious
179aa00a454a97bb1e45e7fb3fb114d9	2017-09-14 07:18:01 UTC	22 of 59 positively flagged this sample as malicious
19b72c2b11d70013fc2147382d75c656	2017-09-14 07:35:39 UTC	21 of 58 positively flagged this sample as malicious
1b52265337ebc39516678869cc2aed5a	2017-09-10 15:00:56 UTC	25 of 57 positively flagged this sample as malicious
1b9ec2551f8ade5f83394b23340ae5c8	2017-10-02 09:44:54 UTC	26 of 58 positively flagged this sample as malicious
1c0bb403ace5a6e2bd6b7409db50d505	2017-08-16 03:45:11 UTC	24 of 57 positively flagged this sample as malicious
1c50bc31a9d27b5cf912c1a2dd73e548	2017-10-03 12:24:40 UTC	29 of 60 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
1cbac18d3bb664473855de7b2b958182	2017-06-20 00:07:25 UTC	5 of 57 positively flagged this sample as malicious
1e3179dbfc95c8bbe0cd33830ae9802c	2017-09-14 06:48:09 UTC	21 of 60 positively flagged this sample as malicious
1f48156c77fa432166b54d5503c1aac2	2017-09-13 04:48:00 UTC	28 of 58 positively flagged this sample as malicious
214307803e25208095b2d27261f088e2	2017-08-24 03:45:52 UTC	24 of 58 positively flagged this sample as malicious
215cc2aa6c9edb33648283cb49da2d99	2017-05-20 21:52:58 UTC	29 of 58 positively flagged this sample as malicious
21652156824d4a074e1b690d4f6bfad7	2015-12-31 11:45:33 UTC	35 of 57 positively flagged this sample as malicious
21aeb76c456e55dc52680da92d11e12d	2017-09-14 05:49:58 UTC	9 of 58 positively flagged this sample as malicious
22b72382ca228ba76e58d9c98236f045	2017-09-14 06:17:50 UTC	20 of 58 positively flagged this sample as malicious
24734ef952fe363415cd4c2f7322276f	2017-10-14 15:06:21 UTC	28 of 60 positively flagged this sample as malicious
25993ee48b86b5a93a47bff5d0d697b8	2017-09-14 06:33:03 UTC	19 of 58 positively flagged this sample as malicious
26dc4799eb1feaa43bec3b0ec3225fee	2017-04-10 18:06:11 UTC	27 of 58 positively flagged this sample as malicious
2733137d5f8a152a2cf50929c0164894	2017-09-14 04:57:20 UTC	21 of 58 positively flagged this sample as malicious
2760b583b79f9b43dbd9aa334b38b6fd	2017-09-11 10:15:11 UTC	28 of 60 positively flagged this sample as malicious
28072a89a50e41ddb7dd9097ba06ee09	2017-09-16 22:14:08 UTC	16 of 59 positively flagged this sample as malicious
2b11b4291193405868a9033fb2c768a1	2016-06-24 18:32:58 UTC	35 of 57 positively flagged this sample as malicious
2ca03ef2125b0335b581302420cb8e91	2017-09-14 05:45:46 UTC	9 of 58 positively flagged this sample as malicious
2cd75d23f526338ac0de7c8bc2fea4ce	2017-07-06 09:34:47 UTC	1 of 58 positively flagged this sample as malicious
2db905373ea58920f7dbf9f3e59ba990	2017-09-10 00:01:08 UTC	14 of 59 positively flagged this sample as malicious
2e912720306afd791206a3784bb743f4	2017-09-14 07:20:11 UTC	16 of 58 positively flagged this sample as malicious
2ed334550bd45ad667ea7d4039ff3bb1	2017-09-14 07:22:27 UTC	23 of 58 positively flagged this sample as malicious
2fa4b143c12b89527b5ad592fbf0692a	2017-06-19 23:43:52 UTC	11 of 57 positively flagged this sample as malicious
2ff923596aa93ab6d03e3e970b5e1198	2017-09-14 07:03:42 UTC	9 of 58 positively flagged this sample as malicious
30362aa28757a76a0e5fd90b81915001	2017-09-14 05:08:21 UTC	25 of 60 positively flagged this sample as malicious
3171681b7e29bcfe85d8f1e2411babcd	2017-09-14 07:21:12 UTC	9 of 57 positively flagged this sample as malicious
32128fa046336d06c328349bce366e1d	2016-12-21 03:39:13 UTC	1 of 55 positively flagged this sample as malicious
33a50a9399f416b125e3302ebd2a132b	2017-09-14 06:27:20 UTC	21 of 58 positively flagged this sample as malicious
34d31584f7e325b0857cc8275b1dd500	2017-09-14 10:10:44 UTC	9 of 59 positively flagged this sample as malicious
35176e86b2e96733188e2f939364117f	2017-09-14 05:11:40 UTC	20 of 57 positively flagged this sample as malicious
35dd1a618443862cda9f77c17aea4ddb	2017-06-19 23:49:43 UTC	6 of 57 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
37cbae5f4249ed569fd3f657de2a36ac	2017-06-19 19:27:07 UTC	6 of 57 positively flagged this sample as malicious
396cbbe2c7ce1f05341ae305621be460	2017-09-14 06:04:18 UTC	27 of 60 positively flagged this sample as malicious
3b3f6977f77c82741d4c1e819d21f670	2017-08-21 19:46:12 UTC	12 of 57 positively flagged this sample as malicious
3e247c19af30e80ac97de97050f41869	2017-06-19 19:26:43 UTC	9 of 57 positively flagged this sample as malicious
3ebfbd542edc4d76a8597d9fbcb3e4c4	2017-06-19 23:56:35 UTC	7 of 57 positively flagged this sample as malicious
40a4b4aebb65d16047e9bf56844ccae3	2017-09-14 07:27:23 UTC	23 of 57 positively flagged this sample as malicious
40f1f759b87035ac6893bd94918d8e7e	2017-05-05 20:50:29 UTC	27 of 57 positively flagged this sample as malicious
417c623a70d8514d888f9179a3bd957e	2017-10-09 12:48:40 UTC	22 of 60 positively flagged this sample as malicious
42743e6af31c9b3a13ac2be41076752e	2017-06-09 21:51:39 UTC	28 of 57 positively flagged this sample as malicious
42db0b662a69a7d94ab3e4f947e7e168	2017-09-14 06:57:49 UTC	18 of 58 positively flagged this sample as malicious
42fe79a930203078da190d7a8e291d3d	2014-09-26 11:29:19 UTC	5 of 55 positively flagged this sample as malicious
43ed5df62f74538552b899ab9c12c08f	2017-09-14 07:14:26 UTC	9 of 58 positively flagged this sample as malicious
44c5badc2a1a145af7e59c2aa9ef6a27	2017-08-17 09:14:12 UTC	25 of 58 positively flagged this sample as malicious
4516f702b804ef767f8719a29f24292d	2017-08-24 09:38:40 UTC	27 of 59 positively flagged this sample as malicious
4716adafa14b337b41a4e14a3200b033	2017-09-14 05:55:43 UTC	20 of 58 positively flagged this sample as malicious
47723d1a7936586ba972838583cc6c9e	2017-09-11 04:37:58 UTC	27 of 59 positively flagged this sample as malicious
483b322b42835227d98f523f9df5c6fc	2016-11-27 11:26:26 UTC	31 of 59 positively flagged this sample as malicious
4a1e830050766ca432536408eca8c58c	2017-06-19 23:56:55 UTC	4 of 57 positively flagged this sample as malicious
4b2620c4d6778087a7ac92aa4cea3026	2017-09-14 05:04:50 UTC	9 of 58 positively flagged this sample as malicious
4c45fc4a7ba1a77b0c7f7479a1036702	2016-06-08 10:08:28 UTC	37 of 58 positively flagged this sample as malicious
4d193825ee038eb1b54c6633678f68e0	2017-09-14 06:36:31 UTC	7 of 56 positively flagged this sample as malicious
4db8073fb6df550e404c6b46efe9f999	2017-06-20 00:07:54 UTC	12 of 57 positively flagged this sample as malicious
4e593af1ab25873681c62ca4f49e31e3	2016-06-16 14:36:44 UTC	35 of 58 positively flagged this sample as malicious
4ef491686122ef9670a3f0925af18d9e	2017-09-14 05:36:11 UTC	24 of 59 positively flagged this sample as malicious
4f5d0ed102de7c171d1df4989c4cdcd0	2016-06-20 05:55:31 UTC	22 of 59 positively flagged this sample as malicious
4f65385b62754f793d9a5e73ef747192	2017-09-14 05:27:58 UTC	21 of 58 positively flagged this sample as malicious
4f8ec335722beb92211c1e87dd736698	2017-09-14 06:27:16 UTC	18 of 57 positively flagged this sample as malicious
50f5f6d1f0f67f15f6a15ffdae671bef	2017-08-15 01:49:35 UTC	25 of 58 positively flagged this sample as malicious
568320b732606052a095f9981f22f811	2017-09-14 06:21:43 UTC	25 of 59 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
5687fcea772c382ec3eba30e7474fbbe	2017-08-23 13:48:17 UTC	26 of 58 positively flagged this sample as malicious
57ae3c3a9341add2e35996231fd4a4d0	2017-09-14 05:45:22 UTC	36 of 60 positively flagged this sample as malicious
57bb3571d1af9aaa5db3d3141a39b3e6	2017-09-14 05:33:40 UTC	21 of 57 positively flagged this sample as malicious
57c514ca4f9c673c346cf448b10d63a4	2017-06-19 23:56:39 UTC	5 of 57 positively flagged this sample as malicious
5803bd08bb5e7243d8f9013a07090e9f	2017-09-14 07:20:17 UTC	21 of 58 positively flagged this sample as malicious
58a2bbdab2aee018609ebe16b4264e36	2017-06-19 19:27:04 UTC	3 of 57 positively flagged this sample as malicious
5a5666fa9a9b7d4bd293508628bd156d	2017-09-14 06:15:14 UTC	28 of 59 positively flagged this sample as malicious
5abdfc799d9df1edae9656b2634e1db9	2016-05-26 09:26:21 UTC	27 of 58 positively flagged this sample as malicious
5b648c78a18b26d037f4b5bff5b8570d	2017-09-14 05:56:22 UTC	9 of 57 positively flagged this sample as malicious
5ba639ecd5618a2bbe5170d768e74919	2017-05-06 16:39:12 UTC	28 of 58 positively flagged this sample as malicious
5bc8cca9ad55d6a64f8e6d4a9ff70515	2017-10-15 17:38:23 UTC	28 of 59 positively flagged this sample as malicious
5d7175a5fadbaa39b8adc4b0d25b6fb3	2017-09-13 08:39:06 UTC	28 of 59 positively flagged this sample as malicious
604309ac21846b22b2caae57bf67f3fb	2014-10-08 23:03:26 UTC	1 of 55 positively flagged this sample as malicious
60f34ddcbc1b17d08fbffaef22b68c54	2017-08-18 13:59:59 UTC	27 of 58 positively flagged this sample as malicious
60fc6ad449a9516e4cc28f90501dcb45	2017-09-13 10:16:50 UTC	24 of 58 positively flagged this sample as malicious
6215e3774235b0198b01591432711b1b	2017-09-14 06:18:17 UTC	20 of 58 positively flagged this sample as malicious
62c041828b1e6912dfb03298ba438a4d	2017-09-14 07:08:25 UTC	20 of 58 positively flagged this sample as malicious
630f3cb8a45c48e705884a3a7a569009	2017-09-14 07:02:52 UTC	21 of 58 positively flagged this sample as malicious
63110ebe3240e9c10f697243c5b20546	2017-08-25 04:59:54 UTC	28 of 58 positively flagged this sample as malicious
631715522c741190a7db60c7a1aa1857	2017-09-14 05:45:17 UTC	20 of 58 positively flagged this sample as malicious
63cb6b921e038f7876ad1df989adae8f	2017-09-11 18:44:58 UTC	31 of 59 positively flagged this sample as malicious
647160e2c4edb1227a9ea7f0515e7802	2017-06-19 23:56:13 UTC	4 of 56 positively flagged this sample as malicious
6600a4555b57717198efa28c2f81a580	2017-01-30 08:23:39 UTC	1 of 54 positively flagged this sample as malicious
66eb028016297b6ae9d83369fc27b8f1	2017-06-20 00:02:04 UTC	10 of 57 positively flagged this sample as malicious
67ab34a6f119169933dde52fbd98449a	2017-09-14 06:38:21 UTC	9 of 57 positively flagged this sample as malicious
68c99433880dcc983856d42bfe89fe18	2017-08-17 07:59:30 UTC	21 of 59 positively flagged this sample as malicious
69485cd1d7f33ee63035b5a51322499d	2017-09-27 14:55:33 UTC	22 of 59 positively flagged this sample as malicious
6bd761f1dc9d89088e32b0cd38a4a0bf	2017-08-31 04:19:29 UTC	29 of 60 positively flagged this sample as malicious
700419e285c8940fb27399b907e5f6f4	2017-09-14 07:55:05 UTC	27 of 59 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
706d02d456accd9f0c595719ecc9e4d7	2017-09-14 05:02:37 UTC	21 of 58 positively flagged this sample as malicious
70e041ceb8cf1649bedde88fcc9f2fe9	2017-09-13 05:12:25 UTC	25 of 58 positively flagged this sample as malicious
70ed42c63f6e928609b4c96c2d9bfed0	2017-09-14 06:54:43 UTC	19 of 58 positively flagged this sample as malicious
725e4daaa2e7871376b8824f081c8407	2017-09-24 15:30:30 UTC	27 of 58 positively flagged this sample as malicious
7266ddd8b30547e7b58be25068c4ca2d	2017-09-14 06:57:51 UTC	27 of 57 positively flagged this sample as malicious
73c64457c379990a2ea9d6727273f153	2017-06-20 00:01:44 UTC	9 of 57 positively flagged this sample as malicious
73d9116e2182ab33cc4ab049e4c184aa	2017-06-19 23:44:14 UTC	5 of 60 positively flagged this sample as malicious
74022ded0c626bc340442eb0b2cde924	2017-09-14 06:31:55 UTC	21 of 58 positively flagged this sample as malicious
7418711b0700bce6c1ec4ba3f73fa7ad	2017-06-19 23:56:11 UTC	5 of 55 positively flagged this sample as malicious
748fe180301f7f36b8f3241a83a90b25	2017-10-03 19:36:27 UTC	27 of 59 positively flagged this sample as malicious
7508cb71dcba0fc3ac0c636baf801fd5	2017-05-01 22:16:34 UTC	25 of 58 positively flagged this sample as malicious
763c1f2b382afaf94e646e9db3d7d0bb	2017-05-12 21:04:05 UTC	28 of 58 positively flagged this sample as malicious
77486750f502a76e530364d2fd7a7571	2017-09-28 01:24:01 UTC	29 of 60 positively flagged this sample as malicious
77fd8616952647a01a3cad7d1ecf93aa	2017-09-09 15:26:01 UTC	27 of 57 positively flagged this sample as malicious
78158b938a3ecfb21ff8aed13482990c	2017-09-11 07:58:18 UTC	28 of 59 positively flagged this sample as malicious
78163c45c6a26741edbbf5517a28401d	2017-08-22 03:52:35 UTC	16 of 57 positively flagged this sample as malicious
79992846a4d5b4e7109aa470bb8b8d26	2017-09-14 07:02:09 UTC	18 of 57 positively flagged this sample as malicious
7a84e11af214468b5095ba3ba499763e	2017-07-08 05:06:43 UTC	25 of 58 positively flagged this sample as malicious
7b06c08d5b89878285412c75e954bc46	2017-06-30 19:46:01 UTC	27 of 57 positively flagged this sample as malicious
7b68d90ee7a225765911ec65535a3470	2017-06-19 19:27:13 UTC	8 of 57 positively flagged this sample as malicious
7bc4166f715cc0c25a9ebadd33bbe3b9	2017-09-26 23:19:33 UTC	22 of 59 positively flagged this sample as malicious
7fb7c97b2e9e0073ea81381289e31263	2017-09-14 07:10:56 UTC	21 of 58 positively flagged this sample as malicious
80967df856279d385c848c588ed551f5	2017-09-14 06:11:47 UTC	25 of 60 positively flagged this sample as malicious
814487db7841e925765f575e1b3020da	2017-09-11 08:12:19 UTC	29 of 59 positively flagged this sample as malicious
81c8f77fe8eab66eb8a160e1e80032b1	2017-09-14 05:52:19 UTC	9 of 58 positively flagged this sample as malicious
832daee7ef733fa06cb2cc6c4dd772e4	2017-08-25 13:39:48 UTC	28 of 57 positively flagged this sample as malicious
845b20c45feb236d4e2660fbe6238ef7	2017-09-14 10:09:20 UTC	25 of 58 positively flagged this sample as malicious
8484ab646e4963979b51c9a743fe813c	2017-09-14 05:55:53 UTC	25 of 56 positively flagged this sample as malicious
849da70b51db35c04df5c4a2b0c49978	2017-09-14 07:33:24 UTC	7 of 58 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
84e3ad0d62d21739d632d2106864e79e	2017-10-14 15:10:00 UTC	20 of 58 positively flagged this sample as malicious
86c1c8fe6a156a44d9af74b23326b1a7	2017-09-14 07:06:54 UTC	7 of 58 positively flagged this sample as malicious
876b3c44516a04af8e7b778a4fb6459c	2017-06-20 00:07:32 UTC	13 of 57 positively flagged this sample as malicious
87955f9b3d487c29f3819534bfb458b8	2017-07-16 01:39:32 UTC	24 of 58 positively flagged this sample as malicious
895c506102e65622d34ec29c864c8e78	2017-09-08 22:55:33 UTC	26 of 58 positively flagged this sample as malicious
89bdece6977230f4a4bf4d9f7bdc450b	2017-06-19 23:57:12 UTC	6 of 57 positively flagged this sample as malicious
8b38f484a0a2e2f1695800ac5867ed0c	2017-09-14 06:46:51 UTC	20 of 57 positively flagged this sample as malicious
8ddb14db9417749384a22cb1ceeb5df5	2017-09-14 06:37:59 UTC	7 of 58 positively flagged this sample as malicious
8e20898079f86f7fea338d0c581dc346	2017-09-14 05:14:47 UTC	9 of 57 positively flagged this sample as malicious
8ec78510a7305d5036b83ea364919329	2017-09-13 10:27:00 UTC	29 of 60 positively flagged this sample as malicious
8f160254d4544759ee2f21ee67e8d499	2017-08-19 19:44:24 UTC	28 of 57 positively flagged this sample as malicious
8f9e3b3bee6284d7d2e60a5e4d380b51	2017-09-14 05:45:20 UTC	9 of 58 positively flagged this sample as malicious
90eb5ae793c603ff5f2bed8405cfda9a	2017-09-14 04:52:16 UTC	18 of 57 positively flagged this sample as malicious
90f4efbebefbb0d7c00fa6d2f3f493ef	2017-09-14 05:40:03 UTC	27 of 59 positively flagged this sample as malicious
9182057f942e294e6411fa09a4e1bc07	2017-09-14 05:18:22 UTC	30 of 58 positively flagged this sample as malicious
92b0647066a4bc5b2354337a3c7e53e1	2017-09-14 06:19:11 UTC	23 of 58 positively flagged this sample as malicious
9590cf63c14047adec7effeaecd50d9a	2017-09-13 10:12:46 UTC	26 of 58 positively flagged this sample as malicious
964cd8930da715979dfbf72ef6542e69	2017-09-11 03:14:01 UTC	24 of 57 positively flagged this sample as malicious
971d5b9a22978c874896f6f4fd55c163	2016-12-21 03:39:03 UTC	2 of 53 positively flagged this sample as malicious
9745d2ee10c917ed2bc5fd2a9b8437ac	2017-06-19 23:56:16 UTC	7 of 56 positively flagged this sample as malicious
97db092615eb0dc51809763ff5543ab5	2017-09-14 04:58:24 UTC	9 of 58 positively flagged this sample as malicious
982f509e3a517985a93584aa60ef6354	2014-03-06 18:10:26 UTC	0 of 56 positively flagged this sample as malicious
984f22e4d7d47e3c4251a9e942a50a88	2017-09-14 05:15:15 UTC	9 of 58 positively flagged this sample as malicious
988ccc200938e8035a706eab1d29f7ad	2017-09-25 17:31:23 UTC	37 of 60 positively flagged this sample as malicious
9a15faa383e018b4373b53635c70ceb2	2017-08-20 17:38:12 UTC	28 of 58 positively flagged this sample as malicious
9d8e3e4c23f6fea431fda602fb00629d	2017-05-06 08:04:04 UTC	30 of 57 positively flagged this sample as malicious
9eba1f4cc856783ef3c3a9d15d221d17	2017-09-14 07:09:41 UTC	7 of 57 positively flagged this sample as malicious
9f2994c909f497f4e2a06acc66da8e9f	2017-09-10 15:14:46 UTC	25 of 59 positively flagged this sample as malicious
a00168464baa118d86c9280c70837dc8	2017-09-25 01:09:32 UTC	16 of 59 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
a058896f22ee796009518eab6a263230	2017-09-14 06:23:37 UTC	9 of 57 positively flagged this sample as malicious
a1d3f07a32b590c449c3ecb105a92bfb	2017-09-14 06:38:20 UTC	20 of 57 positively flagged this sample as malicious
a1d511213200144ea2dcaf440800c6cd	2017-06-19 23:56:19 UTC	4 of 56 positively flagged this sample as malicious
a1f54e3c01df0a94929db5070685c8ad	2017-08-17 01:03:00 UTC	18 of 58 positively flagged this sample as malicious
a4371958b0bf2ef98c4786fc47b271f9	2017-08-23 12:59:58 UTC	25 of 58 positively flagged this sample as malicious
a4944230d62083019d13af861b476f33	2016-06-18 18:32:41 UTC	24 of 60 positively flagged this sample as malicious
a67c1814f5f558b10d11c312b2e2113a	2017-09-27 08:13:54 UTC	15 of 58 positively flagged this sample as malicious
a686d857f840751ba0f6c09387ee2fcd	2017-06-20 00:06:43 UTC	8 of 57 positively flagged this sample as malicious
a6c912cf92592835f9b5a7b0008c72fd	2017-08-20 12:00:06 UTC	25 of 58 positively flagged this sample as malicious
a75f54ecd88370e15929a3c167788650	2017-08-14 06:11:45 UTC	26 of 58 positively flagged this sample as malicious
a7a1abb5d8e87fc670b6841a805103df	2017-04-22 11:00:37 UTC	37 of 59 positively flagged this sample as malicious
a848dd1b189794df9d663875306b5669	2017-08-12 19:18:44 UTC	25 of 58 positively flagged this sample as malicious
a86488274b56159d89203a23060f4d39	2016-06-18 18:32:35 UTC	24 of 60 positively flagged this sample as malicious
a91326d1a79c6e460290a18aa25e021d	2017-08-25 13:25:52 UTC	27 of 58 positively flagged this sample as malicious
a9c23780accb1c2809d4f9a6da0e7ec6	2017-08-29 11:41:45 UTC	29 of 60 positively flagged this sample as malicious
ab4dbede113872843d937b9bb71fd8a7	2017-08-27 08:03:58 UTC	26 of 57 positively flagged this sample as malicious
abb49353283b58ef61f61c76be353f05	2017-10-13 07:10:24 UTC	26 of 59 positively flagged this sample as malicious
ac34800f6312fb3a9667f86887c66bf0	2017-09-14 05:16:23 UTC	21 of 58 positively flagged this sample as malicious
acfb380ad0694dd89dca6a0b81cc2272	2017-06-19 23:56:48 UTC	6 of 57 positively flagged this sample as malicious
ae53acde59e7f0e3a6f4d0d1a6be0ef2	2017-10-05 01:28:11 UTC	26 of 59 positively flagged this sample as malicious
af05768f8b9075c9ae29883c3536653e	2017-09-23 08:44:16 UTC	27 of 59 positively flagged this sample as malicious
af85ff722b21b31701374107f7448cee	2017-08-17 21:10:22 UTC	27 of 58 positively flagged this sample as malicious
afbfedc25605d51346369a98867227b6	2017-09-14 16:47:53 UTC	18 of 59 positively flagged this sample as malicious
b04ce8871e94f850cb1c9c3f74286965	2016-06-20 05:54:34 UTC	27 of 60 positively flagged this sample as malicious
b07745481e11ed4c26d027dee8708a1f	2017-02-26 17:14:34 UTC	29 of 58 positively flagged this sample as malicious
b1e642d300f9e887f3f667e97b26b751	2017-07-08 13:35:10 UTC	23 of 59 positively flagged this sample as malicious
b3d26632c4077e731ef2da329974519d	2017-10-14 15:02:51 UTC	8 of 58 positively flagged this sample as malicious
b414cdc90dc260035dcf2787a534fdde	2017-09-14 06:50:12 UTC	9 of 58 positively flagged this sample as malicious
b74bb1415a46e9e21c36cf688a044186	2017-09-14 07:27:06 UTC	20 of 58 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
b76ffcdafb3861d3c30bb5becb73ec28	2017-09-14 05:23:27 UTC	21 of 60 positively flagged this sample as malicious
b9e84b04d3f9c97912fd4e5e9e7d5346	2017-09-20 22:23:11 UTC	26 of 59 positively flagged this sample as malicious
bee9a7e795527ed632bb42e2ba928363	2017-09-14 05:31:38 UTC	9 of 58 positively flagged this sample as malicious
bef36ad5a5a6b4a5c0dbe5d4cc9c5586	2017-08-15 06:21:54 UTC	27 of 57 positively flagged this sample as malicious
bf0c5d5cfafafc3893c3b4d99f67303c	2017-09-14 07:11:20 UTC	9 of 56 positively flagged this sample as malicious
bf346e8bee16106849bd0f78a004efad	2017-07-06 22:49:30 UTC	1 of 60 positively flagged this sample as malicious
bfb5300d63e8f266f7345b6e32b5bb6b	2017-09-14 06:47:00 UTC	7 of 58 positively flagged this sample as malicious
c0c50b69f325d696a7cdb3311f235500	2017-08-15 02:05:54 UTC	28 of 58 positively flagged this sample as malicious
c10d2e684af1fa079a8229fe3ae45cf5	2017-08-17 16:07:52 UTC	27 of 58 positively flagged this sample as malicious
c13c5b779b9c3e6eaffcdc2addf29942	2017-09-14 06:33:15 UTC	7 of 58 positively flagged this sample as malicious
c1accbab60a70d1b20b7fde2c73c5d76	2017-06-20 00:01:47 UTC	5 of 57 positively flagged this sample as malicious
c2e67c8380ae5545e505cd44df4c702a	2017-08-18 12:43:18 UTC	27 of 58 positively flagged this sample as malicious
c3d6bff74f0c40ccb3197c4f6f71e6eb	2017-09-13 08:14:42 UTC	25 of 58 positively flagged this sample as malicious
c535fa75588dfa2c5c1b8c4c4473774c	2017-08-15 01:15:07 UTC	26 of 58 positively flagged this sample as malicious
c6cf74ca4d29ebbadb876394922acda0	2017-09-14 05:59:55 UTC	26 of 60 positively flagged this sample as malicious
c81f2d82a2809f7c576021e63d3f727c	2017-08-23 22:44:21 UTC	27 of 58 positively flagged this sample as malicious
c83e26d778d5bf5b21861c75fdabb48d	2017-09-14 05:27:50 UTC	9 of 58 positively flagged this sample as malicious
c90561275cdac5b734052f87cf9ff38e	2017-08-18 14:13:20 UTC	26 of 58 positively flagged this sample as malicious
caa689187bf47e5fd2a2657cec0df6d5	2017-09-26 07:42:04 UTC	26 of 58 positively flagged this sample as malicious
cac6603b4e6dab11c66581d89383a27c	2017-09-14 06:51:17 UTC	21 of 58 positively flagged this sample as malicious
cb978527dc707aaa98504f14e58df5a6	2017-09-14 06:26:18 UTC	9 of 58 positively flagged this sample as malicious
cb9f5a1898f96b7d8efcd18ec6e13f07	2017-09-14 05:54:37 UTC	20 of 56 positively flagged this sample as malicious
cc064f8f4f8fe15f8d7fc07453ab8ee4	2017-09-26 01:50:11 UTC	29 of 60 positively flagged this sample as malicious
ccb487179fe72da2c47e58eee380a260	2017-09-14 06:36:44 UTC	20 of 57 positively flagged this sample as malicious
ce31c046270623f3fd157a882449b53f	2017-09-14 07:31:52 UTC	7 of 57 positively flagged this sample as malicious
ce735a1a4202176505df4f5cd9ff4a0a	2017-09-14 05:36:25 UTC	17 of 58 positively flagged this sample as malicious
d308b9b4d4f70b95003b23e3ada307bd	2017-09-14 04:58:31 UTC	9 of 58 positively flagged this sample as malicious
d5e40f3e2d31e6c6c00d715a028db5bf	2016-06-15 09:14:46 UTC	36 of 58 positively flagged this sample as malicious
d630c62215c2cc468450fd3b578c8a45	2017-04-26 12:22:06 UTC	25 of 56 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
d8badd195f857f9cb0ecaf86ed6d32fd	2017-10-13 07:35:08 UTC	36 of 59 positively flagged this sample as malicious
d91b28fc92246ac0ac0ab45bb814a586	2017-09-14 05:12:53 UTC	9 of 58 positively flagged this sample as malicious
da0a7b5ade941f44c2a254444bf8f6f6	2017-09-14 05:33:20 UTC	21 of 58 positively flagged this sample as malicious
dc0000195aa0fe2d3f8a6a977fb72a5d	2017-09-13 12:32:31 UTC	19 of 58 positively flagged this sample as malicious
dc4a890cb15d3ce37fb5ed81d2db8d0b	2017-06-19 23:57:57 UTC	4 of 56 positively flagged this sample as malicious
dd512bf7255bcfe3f1aeb1bfd2395cba	2017-09-14 07:03:14 UTC	20 of 58 positively flagged this sample as malicious
de7a309e7288b276fa5e17dff62d5350	2017-05-18 02:26:43 UTC	25 of 57 positively flagged this sample as malicious
df08353fe242893b11fcb14b4315b264	2017-09-14 07:30:52 UTC	16 of 58 positively flagged this sample as malicious
e2bd3ead1d36071c0b7b3192535a9a8f	2017-09-14 05:06:20 UTC	20 of 58 positively flagged this sample as malicious
e47961f9c406d31eab55e8d96802bef8	2017-10-14 19:29:29 UTC	26 of 60 positively flagged this sample as malicious
e545bfb8dc484bd394d87dc5f9d908c3	2017-09-10 18:10:39 UTC	28 of 60 positively flagged this sample as malicious
e56fd7e8979edecf2e5a60c736e0d682	2016-12-21 03:39:11 UTC	1 of 54 positively flagged this sample as malicious
e631a27538a0731e2fec247f76d5987e	2017-09-14 05:14:27 UTC	28 of 58 positively flagged this sample as malicious
e7355da37408a07ef759fc48bbfdfe7e	2017-09-14 02:58:36 UTC	37 of 60 positively flagged this sample as malicious
e73db2a8d719529cdc28bc66c430904b	2017-05-28 22:23:46 UTC	26 of 58 positively flagged this sample as malicious
e8866f7f63d608b19268473db8b8fd90	2017-09-14 05:32:03 UTC	21 of 58 positively flagged this sample as malicious
e912b098f5a8e021e9e6d583cc34dd6e	2017-01-30 08:23:30 UTC	1 of 55 positively flagged this sample as malicious
e9315e0769af400d495a7de50ccf54e2	2017-09-14 05:25:15 UTC	18 of 57 positively flagged this sample as malicious
eb3b90511541039998528348722 57d0d	2017-08-20 01:56:07 UTC	26 of 58 positively flagged this sample as malicious
eb6f16478b50df8d0f479eb47c7c557c	2017-08-20 20:37:27 UTC	26 of 58 positively flagged this sample as malicious
ec41436988e3356ce8b93c5803aa7e6c	2017-09-14 07:33:10 UTC	19 of 58 positively flagged this sample as malicious
ee25b5aecaa22190352bf59287f29161	2017-09-14 05:52:17 UTC	21 of 58 positively flagged this sample as malicious
f05c16b6fdfe3b1e099352c3d8002aa7	2017-09-14 05:58:46 UTC	9 of 56 positively flagged this sample as malicious
f11454c3ff0614432dc2cabb8a012656	2017-09-14 06:44:01 UTC	16 of 57 positively flagged this sample as malicious
f1de31cada16698cc6d212bc0f5db06d	2017-09-14 05:04:58 UTC	25 of 58 positively flagged this sample as malicious
f3b04da9a52b547533399244efd24f55	2017-08-25 13:59:18 UTC	28 of 60 positively flagged this sample as malicious
f3e1f5db377c2f11e25cbd2aa9343d37	2016-06-20 05:54:24 UTC	26 of 60 positively flagged this sample as malicious
f43e971c37d492191ee973c42d7decc2	2017-06-19 23:56:45 UTC	5 of 57 positively flagged this sample as malicious
f477afa7cafc0f8f1bf563262a96519a	2017-09-14 05:46:19 UTC	9 of 58 positively flagged this sample as malicious

MD5	Date of first submission	Scan Summary
f56b750439eb42693334ae0c330461d4	2016-12-21 03:39:08 UTC	1 of 55 positively flagged this sample as malicious
f5dd74f08f9ea90aaeda2b2c43ea9859	2017-09-11 11:08:33 UTC	15 of 59 positively flagged this sample as malicious
f5e6002beb92d913a1ee8fdbad4eaac1	2017-09-27 20:40:22 UTC	24 of 60 positively flagged this sample as malicious
f6fdb413695acc50a536cec329214174	2017-09-14 05:26:35 UTC	19 of 58 positively flagged this sample as malicious
fde76bf4faeb9316127260c1f4b6142f	2017-07-07 17:12:45 UTC	25 of 58 positively flagged this sample as malicious
fdf19272e88f012e17b997f717e1b6d7	2017-09-14 06:01:05 UTC	9 of 58 positively flagged this sample as malicious
fe06e8e7a59cc6a50925a335f6e9c3fa	2017-09-14 04:57:36 UTC	9 of 58 positively flagged this sample as malicious
fe82c7fbcac1b1868a3c8401ea906bf1	2017-04-06 12:13:27 UTC	20 of 60 positively flagged this sample as malicious
fde04f4492b96f449fd36fe10c0e9f3c	2018-03-23 07:58:41 UTC	15 of 59 positively flagged this sample as malicious
35b6e58366611f17781a4948f77353b6	2018-03-23 07:55:33 UTC	23 of 59 positively flagged this sample as malicious
f6b9127970d56de9a65419cb628206af	2018-03-02 19:09:25 UTC	17 of 60 positively flagged this sample as malicious
eb8887024deb1889b5ac6cee37c9ef7d	2018-03-02 19:12:29 UTC	24 of 60 positively flagged this sample as malicious