Effectiveness of Linux Rootkit Detection Tools

University of Oulu
Faculty of Information Technology and Electrical Engineering
Degree Programme in Information Processing Sciences
Master's Thesis
Juho Junnila
27.3.2020

Abstract

Rootkits – a type of software that specializes in hiding entities in computer systems while enabling continuous control or access to it – are particularly difficult to detect compared to other kinds of software. Various tools exist for detecting rootkits, utilizing a wide variety of detection techniques and mechanisms. However, the effectiveness of such tools is not well established, especially in contemporary academic research and in the context of the Linux operating system.

This study carried out an empirical evaluation of the effectiveness of five tools with capabilities to detect Linux rootkits: OSSEC, AIDE, Rootkit Hunter, Chkrootkit and LKRG. The effectiveness of each tool was tested by injecting 15 publicly available rootkits in individual detection tests in virtual machines running Ubuntu 16.04, executing the detection tool and capturing its results for analysis. A total of 75 detection tests were performed.

The results showed that only 37.3% of the detection tests provided any indication of a rootkit infection or suspicious system behaviour, with the rest failing to provide any signs of anomalous behaviour. However, combining the findings of multiple detection tools increased the overall detection rate to 93.3%, as all but a single rootkit were discovered by at least one tool. Variation was observed in the effectiveness of the detection tools, with detection rates ranging from 13.3% to 53.3%. Variation in detection effectiveness was also found between categories of rootkits, as the overall detection rate was 46.7% for user mode rootkits and 31.1% for kernel mode rootkits. Overall, the findings showed that while an individual detection tool‘s effectiveness can be lacking, using a combination of tools considerably increased the likelihood of a successful detection.

Keywords
rootkits, rootkit detection, digital forensics, computer security, operating systems, Linux

Supervisor
Ph.D., University lecturer, Ari Vesanen

Foreword

This study was born of a personal curiosity to develop a deeper understanding of how rootkits work and how they can be detected using freely available tools. Its origins can be traced back to Trammell Hudson's exceptional presentation of the Thunderstrike EFI rootkit at the 31st Chaos Communication Congress (31C3) conference, which profoundly impressed me with its exotic nature, technical prowess and compelling narrative. Narrowing the research context to Linux was also due to both personal and professional interests, as it is the operating system I'm most deeply familiar with and the one I use and value the most.

I would like to thank my thesis supervisor Ari Vesanen for his adept guidance and for the intriguing discussions during our meetings. I would also like to extend my sincere gratitude to my family and friends for all their support and patience during my work on this thesis, my studies and everything else.

Juho Junnila

Oulu, March 27, 2020

Abbreviations

AIDE: Advanced Intrusion Detection Environment

AMT: Active Management Technology

APT: Advanced Package Tool

EFI: Extensible Firmware Interface

HIDS: Host-based Intrusion Detection System

HPC: Hardware Performance Counter

IDT: Interrupt Descriptor Table

LKM: Loadable Kernel Module

LKRG: Linux Kernel Runtime Guard

ME: Intel Management Engine

OSSEC: Open Source HIDS Security

PAM: Pluggable Authentication Modules

PID: Process ID

SMM: System Management Mode

SSDT: System Service Dispatch Table

UID: User ID

UMH: Linux Usermode Helper

VM: Virtual Machine

VMI: Virtual Machine Introspection

VMM: Virtual Machine Monitor

Contents

1. Introduction

This chapter describes the background and motivation for this study. The research problem and research questions are also defined. Finally, the utilized research method and the structure of the thesis are described.

1.1 Motivation

A rootkit is a type of software that specializes in hiding its presence in computer systems. The main purpose of a rootkit is to facilitate continuous access to the system it is installed on. While often associated with malware, the use of rootkits is not necessarily limited to malicious purposes, as they can also be employed for legitimate functions such as remote system management and advanced anti-malware solutions utilizing stealth. (Hoglund & Butler, 2006, p. 4)

Regardless of the legitimacy and underlying purposes of rootkits, specific tools can be utilized by computer forensic investigators, information security professionals, system administrators and other users to assist in checking that no unauthorized, continuous access to computer systems is taking place. As such, rootkit detection can be a useful tool for assisting in the process of inspecting a computer system's state of security as a part of a more comprehensive information assurance strategy, or even just for satisfying personal curiosity.

Because rootkits strive to hide themselves and/or other software, detecting them may require specialized tools and techniques. While memory forensics and other methods of manual analysis can provide a more flexible and effective approach to rootkit detection, employing them generally requires more knowledge and resources, especially time (Bunten, 2004; Wampler & Graham, 2007). In contrast, automated detection tools may be utilized by users with less knowledge of operating system internals and concepts of information security in general (Freiling & Schwittay, 2007). This suggests that rootkit detection tools can be relevant for continuous reactive system monitoring and in scenarios where no applicable expertise or resources are readily available.

1.2 Research problem and questions

The effectiveness of detecting modern Linux rootkits using rootkit detection tools is not currently clearly understood. This is evidenced by the small number of studies conducted on the topic, especially in recent years. There is also a lack of substantial, up-to-date empirical evidence on how publicly available detection tools and, by extension, the techniques they utilize compare against one another in the context of detection effectiveness. In addition, it is not entirely clear how effective the tools are in detecting different categories of rootkits. To fill this research gap, the objective of this study was to empirically determine how effective selected Linux rootkit detection tools are in detecting a selected set of currently prevalent publicly available rootkits targeting Linux-based systems.

In an effort to find answers to the research problem, the following research questions were addressed in this study:

RQ1: Can rootkit detection tools be used to effectively detect modern Linux rootkits?

RQ2: What differences are there in the effectiveness of rootkit detection tools for Linux?

RQ3: What kinds of rootkits can be effectively detected using rootkit detection tools for Linux?

1.3 Research method

An experimental research combining a trial and comparison experiment (Tedre, 2014) was carried out in an emulated environment by infecting virtual machines running Linux distributions with chosen rootkits. The selected detection tools were then used to attempt to discover indications of the rootkits' presence in the system. Data on whether a detection tool was able to detect a rootkit or not was collected to facilitate further analysis and evaluation of effectiveness.

The effectiveness of each detection tool was evaluated by determining whether the tool was able to successfully detect a rootkit, provide a warning of a possible infection/compromise or suspicious behaviour, report a false positive or miss the rootkit's existence entirely. The results were analysed to produce a comparison of each tool's effectiveness in detecting a specific rootkit. This took into consideration the number of successful detections, false negatives as well as false positives. Based on the analysis, detection rates were calculated for each detection tool. The analysis was then used to answer the research questions and summarize the findings.

1.4 Structure

This study is structured as follows: Chapter two discusses the theory relating to rootkits and rootkit detection, including prior research conducted on the topic. Chapter three describes the research method, environment and tools used as well as the procedure for evaluation. Chapter four describes the empirical results and findings of the study in detail. Chapter five summarizes the findings and provides discussion on their meaning, implications and relation to prior research. Finally, Chapter six concludes the study, summarizing its contents as well as presenting the limitations of the findings and proposing future research directions.

2. Background

This chapter describes the literature search conducted to find relevant studies related to this study's topic, as well as the prior research that is directly related to this study. In addition, key concepts about rootkits, rootkit detection and some Linux operating system concepts relevant for the topic are described.

2.1 Literature search

In order to find relevant sources for this study, a search for related literature was conducted. Three academic search engines were used: Google Scholar, Microsoft Academic and CiteSeerX. In addition, five full-text library databases were queried: IEEE Xplore, ACM Digital Library, Wiley Online Library, arXiv E-print Archive and SpringerLink.

Terms used for the search included the names of the rootkits and rootkit detectors involved in this study. Various terms and synonyms for rootkit detectors were also applied. Generic keywords such as names of operating systems and terms describing effectiveness as well as evaluations were used in combination with the other terms in an attempt to find literature related to the evaluation of rootkit detection effectiveness in any context. Any paper with a title considering the topic of rootkits or rootkit detection was regarded as relevant. For detailed search terms and corresponding result counts, see Appendix A.

For Google Scholar, about 3200 results were returned for the search, around 300 of which were deemed to be relevant for this study's topic, based on a cursory review of the papers' titles. For Microsoft Academic, about 3000 results were returned, around 300 of which were deemed relevant. For CiteSeerX, about 8500 results were returned, about 100 of which were considered relevant. The results were sorted by relevance and the titles were sequentially inspected until a full page (of 20 results) no longer referenced rootkits or rootkit detection in the papers' titles.

For IEEE Xplore, 130 results were returned, 30 of which were relevant. For ACM Digital Library, 114 results were returned, 18 of which relevant. For Wiley Online Library, 15 results were returned, only 3 of which were considered relevant. For arXiv E-print Archive, 18 results were returned, 7 of which relevant. Finally, 351 results were returned for SpringerLink, 41 of which were found to be relevant for the topic.

Most papers referenced in this study were obtained via Google Scholar, IEEE Xplore, ACM Digital Library or SpringerLink. Both IEEE Xplore and ACM Digital Library were found to host multiple relevant conference and journal publications on the topic of computer security, digital forensics and operating systems, while the Lecture Notes in Computer Science series available via SpringerLink was found to include several relevant conference proceedings on digital forensics and intrusion detection.

2.2 Evaluation of rootkit detection effectiveness

Not much prior research is available on rootkit detection and the effectiveness of specific detection methods, even less so in the context of Linux rootkits. What research exists on the topic is either unavailable online or quite dated, with most of it conducted before the 2010s. However, existing research does provide some valuable information on the obfuscation techniques used by rootkits as well as how they can potentially be detected. Specifically, papers by Bunten (2004), Todd et al. (2007) as well as Freiling and Schwittay (2007) describe particularly relevant evaluations of the effectiveness of rootkit detection methods.

An overview of rootkits for Unix-based systems (including Linux) has been presented, describing some of the techniques utilized by them as well as presenting methods available for detecting and preventing them. The overview categorized Unix-based rootkits into two categories (user mode and kernel rootkits) and posited that rootkits were first operating in user mode but have since been largely eclipsed by kernel rootkits, which can modify the behaviour of the operating system kernel and thus potentially gaining greater control of the system. (Bunten, 2004.)

Three methods for preventing kernel rootkits from operating have also been presented in the same study. The first is disallowing the loading of kernel modules at runtime. The second is using mandatory access controls (MAC) to prevent unauthorized access to the kernel's virtual memory via the `/dev/kmem` device file. The third is utilizing kernel modules that specifically look for rootkits and attempt to prevent them from executing in the first place. (Bunten, 2004.)

File integrity tools such as AIDE and Tripwire, rootkit detectors such as Chkrootkit and Rootkit Hunter as well as kernel integrity checkers such as KSTAT have been highlighted as prevalent rootkit detection methods. In addition, manual methods such as measuring changes in the number of instructions executed by system calls, analyzing logfiles and oddities in the file system (especially in `/proc`), running intrusion detection systems as well as conducting forensic analysis on the contents of memory dumps have been discussed. (Bunten, 2004.)

An evaluation of some of the rootkit detection methods' effectiveness against a small set of Linux kernel rootkits (Adore, Adore-NG and SucKIT) has also been conducted, where from a total of 15 detection attempts 10 were successful, with Chkrootkit and AIDE finding all three rootkits. However, the conclusion of the evaluation was that more universal detection methods are required to counter the obfuscation techniques of modern rootkits and that general runtime detection methods are in need as well, potentially utilizing methods such as kernel execution flow analysis and integrity checking. (Bunten, 2004.)

An evaluation of rootkit detection effectiveness similar to Bunten (2004) has been described, with the distinction that the focus of the evaluation's research was on Windows rootkits and determining how much information detection methods can obtain from rootkits. The evaluation considered five rootkits (two user mode and three kernel rootkits) with detection attempts by four rootkit detectors, four "offline analysis tools" (two anti-virus and forensic investigation tools) and a single "live analysis tool",

executing each detection tool in a virtual machine running Windows XP infected with one rootkit at a time. (Todd et al., 2007.)

The results of the evaluation showed that rootkit detectors provide the most useful and plentiful information of the three detection tool types, being able to clearly detect rootkits in 14 out of the total of 25 cases, in addition to six partial detections indicating a hidden process. The authors noted that there are differences in the tools' detection abilities, as two could only detect hidden files while the other two could also detect hidden drivers, ports, processes and services. (Todd et al., 2007.)

In the same evaluation study, a case of a "network defense competition" using the paper's methodologies was presented. The competition targeted eight systems: two virtual machines running Windows Server 2003, two running Fedora Core 2 and four running Windows XP, each with a single rootkit installed. The competitors were unaware of what the security status was for each system. The case featured the use of two anti-virus tools and three rootkit detectors for Windows as well as two rootkit detectors for Linux (Chkrootkit and Rootkit Hunter). The results showed that the rootkit detectors fared well in the less controlled competition environment as well, with claims that rootkit detectors are the leading investigative tools for rootkit forensics. (Todd et al., 2007.)

Another experimental evaluation of the effectiveness of rootkit detection on Windows has been conducted. The evaluation considered 11 detection tools and 9 rootkits in four different versions of Windows. The effectiveness of each detection tools was compared by assigning each a score based on their detection results; a tool was given a score of 0 if it failed to detect a rootkit, 1 if it detected some modifications made by the rootkit or 2 if it detected all modifications made by the rootkit. Two tools were able to detect all but one rootkit, while the other tools provided mixed results. The results prompted the researchers to propose a detection methodology where three detection tools (BlackLight, IceSword and SVV) are used in combination for reliable rootkit detection in different contexts and investigator skill levels. The research advocated for similar experiments to be conducted in the future to verify the detection rates of rootkit detection tools. (Freiling & Schwittay, 2007.)

A process for experimentally evaluating the effectiveness of a rootkit detection tool implantation has also been described. The evaluation considers a view comparison (cross-view-based) detector using virtual machine introspection (VMI). The evaluation data gathering process consists of creating a virtual machine, defining a clean state without rootkits injected, executing the tool with the clean state, injecting a rootkit, executing the tool again and extracting the tool's results for analysis. This process is repeated for each rootkit sample. The tool's effectiveness is evaluated by comparing results of the test run against the clean state and states where rootkits are injected, in addition to checking that the results match the behaviour specified for each rootkit through reference information. (Richer, Neale & Osborne, 2015.)

2.3 Linux

There are a few concepts in the Linux operating system and its user-space tools that are relevant for understanding how rootkits are injected and how they manage to operate in the system. The concepts of library preloading with dynamic linking, separation of the user and kernel space, system call handling and loading code to the kernel using kernel modules will be briefly explained. It should be noted that there are many other key

concepts related to rootkit injection and operation which are not described here, as doing so would be disproportionate for the scope of this study.

Dynamic linking is a method for linking libraries to programs at runtime, where libraries can be shared by the programs that require them. This is distinguished from static linking, where linking happens at compile time. In Linux, it is possible to use the `LD_PRELOAD` environment variable to specify a shared object whose function implementations will be loaded before and preferred over other libraries for a program's execution. (Free Software Foundation, 2015.)

For example, the C standard library function `fopen()` could be defined in a custom shared object to change its behaviour (e.g. return a file pointer to `/dev/zero` whenever attempting to open `/dev/urandom`, otherwise return a pointer to the actual requested file) when called by a program loaded with the `LD_PRELOAD` variable set to the path of the shared object. The `LD_PRELOAD` variable is one of two common ways of injecting rootkits into programs, the other being the use of a global preload configuration file (Carbone, 2014).

While `LD_PRELOAD` is applicable to a shell session or single executables, library preloading can also be achieved for all user-space executables. This type of global library preloading utilizes the `/etc/ld.so.preload` file. The file may contain a list of paths to preloadable shared libraries that are loaded before other shared libraries whenever the dynamic loader is invoked. (Free Software Foundation, 2015.)

The Linux kernel operates in its own part of virtual memory called the kernel space, which is separated from the user space used by applications. Only the kernel (and its modules) can directly access hardware and system resources, so it has absolute control over the system. Processes executing in the user space can request resources or services from the kernel via system calls, which transfer execution to the kernel using either the `SYSCALL/SYSENTER` instruction (in x86-64 after kernel version 2.6) or an interrupt (in x86). The kernel handles system calls by passing execution to the system call handler pointed to by the system call table, using the appropriate system call number as its index. Finally, the system call is handled by the kernel and execution returns to the process that initiated the system call, unless an error occurs. (Love, 2010, pp. 69-83.) The normal flow of handling a system call, in this case `open()`, is illustrated in Figure 1.

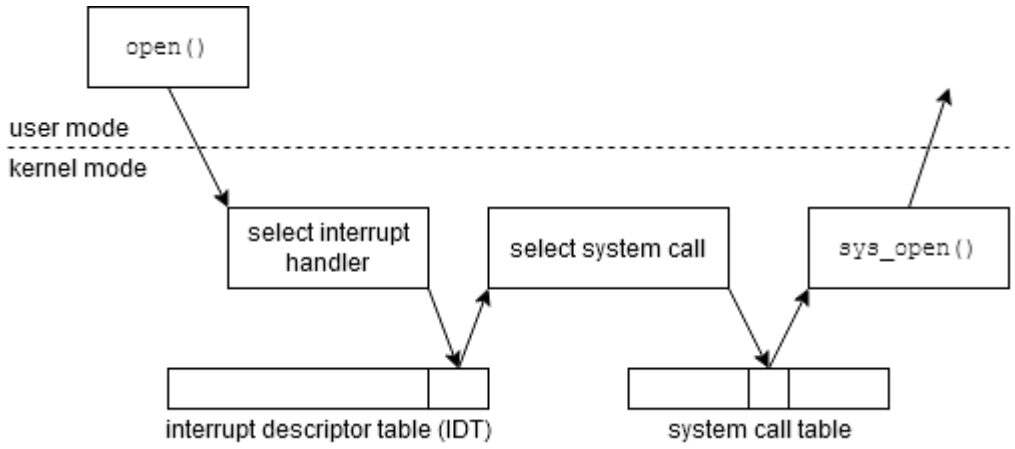

Figure 1. System call handling by the Linux kernel (adapted from Bunten, 2004).

Manipulating the system call table is one of the key methods used by rootkits operating in kernel space to infect an operating system and control the kernel's behaviour. However, many other kernel data structures can also be used for injecting rootkits into the kernel space. These include interrupt handlers, exception tables, debug registers and function trampolines. (O'Neill, 2016, p. 225.)

The functionality of the Linux kernel can be dynamically extended with loadable kernel modules (LKM), which are generally used to load drivers and modify the kernel's behaviour while it is running. LKMs can call kernel functions and thus alter its data structures, making them a powerful tool for manipulating the kernel. (Henderson, 2006.) LKMs are a very common technique for implementing kernel mode rootkits, as they can be easily loaded and don't require the kernel to be recompiled, making it simpler and easier to inject a rootkit into the kernel (Bierfert, 2007). Loading LKMs can be disabled or restricted in the kernel to limit the attack surface for kernel rootkit injection, though this may prevent the system from working as expected, especially when external hardware is connected to the system (Bunten, 2004).

2.4 Rootkits

There exist various definitions of what rootkits are, with a varying degree of emphasis on their intended use. Some definitions classify rootkits as malware used for nefarious purposes while others simply specify them as a type of software with a single purpose: to hide the presence of some software (often themselves) in the operating system they're installed on. Therefore, it is relevant to specify exactly what definition is used in this study.

Russinovich (2006a) has defined a rootkit as "software that hides itself or other objects, such as files, processes, and registry keys, from view of standard diagnostic, administrative, and security software". Russinovich's definition is supported by a similar description: "a rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer" (Hoglund & Butler, 2006, p. 4). A more comprehensive definition describes a rootkit as a set of software tools installed to a system with the purpose of allowing continuous, privileged and arbitrary control and access to the system as well as disguising or controlling access to its files, processes, ports and other objects (Harley & Lee, 2007).

Often installed by attackers for the purpose of leaving a persistent yet hidden method of access to a computer system, rootkits are frequently associated with malicious purposes. However, the definition by Harley and Lee (2007) does not strictly preclude rootkits from being used for legitimate applications and is a sufficiently appropriate generalized definition. It is therefore the definition used in this study.

Rootkits can be categorized based on which privilege level they execute at. A common practice is to divide rootkits into two categories: user mode and kernel mode (also referred to as user-level and kernel-level), as done by Bunten (2004), Harley and Lee (2007), Todd et al. (2007) and many others. These two categories effectively specify whether the rootkit executes in the user space or the kernel space of the operating system.

User mode rootkits execute in the user space of the operating system, and thus do not have direct access to the kernel. Some early rootkits, such as t0rnkit, overwrite key programs like `ls`, `ps`, `login` and `top` with custom code which filters the output of the

programs based on rules specified by the attacker. To hide traces of intrusion, such rootkits may also manipulate logfiles, though they often still leave behind some indications of system intrusion. (Bunten, 2004.)

User mode rootkits can hook into executables by modifying or overriding functions in dynamically linked libraries (Carbone, 2014). Linux-based user mode rootkits can leverage the preload mechanism in the dynamic linker/loader to intercept calls to library functions and control their execution in various ways. (Tian, Wang, Zhou & Zhang, 2011) One example of a known modern Linux user mode rootkit is JynxKit (ErrProne, 2011a), which can open an SSL-encrypted backdoor and hide files and directories by hooking functions in the C standard library such as `open()` and `readdir()`. The rootkit also uses the preload mechanism for injection. Figure 2 illustrates the hooking of `open()` by JynxKit for the `ls` command to hide specific files (matching predefined variables `MAGIC_GID` or `MAGIC_DIR`) when injected with the preload mechanism.

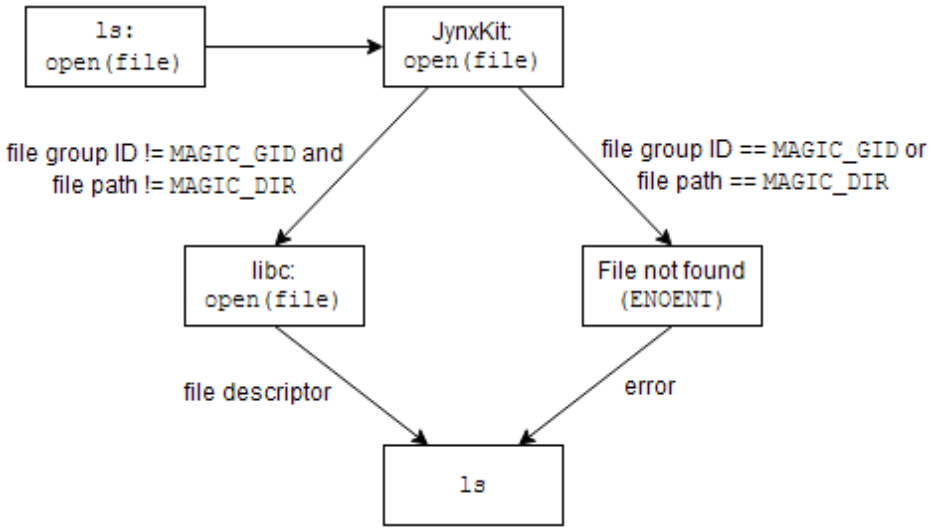

Figure 2. Hooking of `open()` by JynxKit (adapted from JynxKit source code, ErrProne, 2011a).

Kernel mode rootkits operate in the kernel space, which gives them more privileges than can be acquired by user mode rootkits in the user space (Bravo & Garcia, 2011). Such rootkits can directly intercept and override system calls to alter the operating system's behavior to exert deep control over the system (Bunten, 2004). In Linux, there are a few ways kernel mode rootkits can be attached to the kernel. A kernel mode rootkit can be loaded as an LKM, which is allowed to modify many kernel data structures (such as the system call table) via the interrupt descriptor table (IDT) (Tian, Wang, Zhou & Zhang, 2011). It can also be injected using the `/dev/kmem` device file to write the rootkit code to the kernel's virtual memory space and then executing it as a system call (Bunten, 2004). However, more modern kernel versions no longer allow modifying the kernel memory by writing such files as a security measure (O'Neill, 2016, p. 229).

As with user mode rootkits, kernel mode rootkits generally rely on hooking, though in their case a common target is the operating system's system call table. The rootkits can manipulate the interrupt handler routines, IDT and the system call table so that whenever a system call is called, the execution flow is directed to the rootkit's code. The rootkit can then use the original system call on its own discretion when needed. (Tian, Wang, Zhou & Zhang, 2011.)

For example, the SucKIT kernel mode rootkit for Linux creates a copy of the kernel's system call table and changes many of the copied table's entries to point to the rootkit's code, while also modifying the interrupt handler to point to the modified system call table. Additionally, the rootkit overrides the `init` daemon with its own implementation on system boot while redirecting access to `/sbin/init` to the original `init` so that any checksum calculated from it matches the original unmodified version, foiling some simple detection attempts. (Bunten, 2004.) The system call table manipulation conducted by SucKIT is visualized in Figure 3, representing a common operating mechanism for kernel mode rootkits. This is contrasted with the normal system call handling shown in Figure 1.

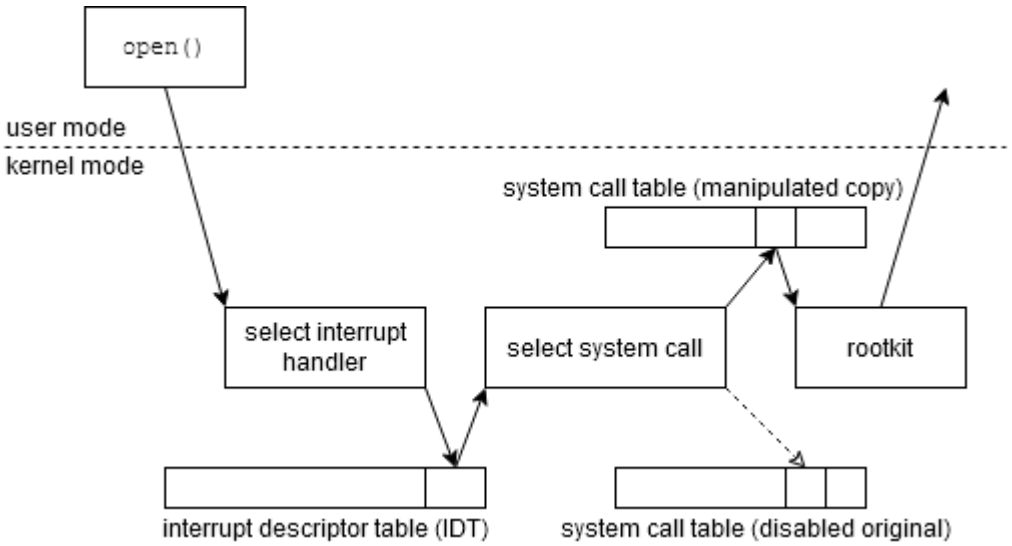

Figure 3. System call table manipulation by the SucKIT rootkit (adapted from Bunten, 2004).

In the Windows operating system family, kernel rootkits can hook into kernel data structures such as the system service dispatch table (SSDT), IDT and individual kernel functions. They can also be placed within layered driver stacks in the Windows Driver Model framework to intercept and manipulate data passed between the drivers. (Kim, Park, Lee, You & Yim, 2012.) However, these mechanisms are not covered as this study purposely focuses on Linux rootkits.

In the interest of completeness, it should be noted that while the two rootkit categories are sufficient in distinguishing the rootkits examined in this study, they are not extensive enough to categorize all types of rootkits in existence. Rootkits may also often exist in multiple categories, as they can operate at various privilege levels and protection rings at once. This gives them different levels of control and access to systems.

Besides user mode and kernel mode rootkits, there exists a variety of rootkits with the capability of operating beyond the kernel's immediate reach, such as bootkits as well as firmware and hardware rootkits. Such rootkits can resist attempts to detect and remove them by operating at levels with more privileges than the kernel itself. As an example of a low-level rootkit, the Thunderstrike EFI rootkit (Hudson & Rudolph, 2015) modifies the boot ROM of a wide range of Apple's MacBook series of laptops.

Infecting the device before the operating system has been loaded allows Thunderstrike to survive any software-based removal attempts, including re-installation of the operating system and even replacement of the device's storage devices. The rootkit,

however, requires a separate device to inject itself into a target system. (Hudson & Rudolph, 2015.) An improved version, Thunderstrike 2, does not require any hardware for injection and can in fact infect peripheral devices such as Thunderbolt adapters to spread to other devices as well (Hudson, Kovah & Kallenberg, 2015).

Table 1 shows a categorization of protection rings (a concretization of privilege levels), with an example of a rootkit existing at each privilege level. A lower ring value indicates greater privileges. The rootkits used for this study operate at rings three (user mode) and zero (kernel mode).

Table 1. Protection rings (adapted from Tereshkin & Wojtczuk, 2009).

Protection ring	Domain	Example rootkit
Ring 3	User mode	JynxKit2
Ring 0	Kernel mode	SucKIT
Ring -1	Hypervisor	Blue Pill
Ring -2	System Management Mode (SMM)	The Watcher
Ring -3	Chipset/firmware (e.g. ME, AMT)	Thunderstrike

It should be noted that there is no universally established consensus on the classification of rootkits in the negative protection rings. This is especially true for "ring -3" rootkits as described by Tereshkin and Wojtczuk (2009), who place them at the chipset level. Potential targets for chipset level rootkits include the Intel Management Engine (ME) and Active Management Technology (AMT) administration systems, which are run directly on a separate chipset, independent of the computer's CPU (Tereshkin & Wojtczuk, 2009).

2.5 Rootkit detection

Rootkit detectors can be categorized into five distinct categories: signature-, behaviour-, cross-view-, integrity-, and hardware-based detectors (Todd et al., 2007). Each category has a different set of mechanisms with advantages and disadvantages inherent to them, and it's not uncommon for rootkit detection tools to utilize techniques from multiple categories. For example, the Rootkit Hunter (Rootkit Hunter, 2018) rootkit detector incorporates signature, behaviour and integrity checking in its suite of detection mechanisms.

Signature-based detectors check files for data that is characteristic of certain rootkits, such as specific strings stored in files or binaries (Todd et al., 2007). Many signature-based rootkit detectors rely on a database of such signatures, indicative of the presence of known rootkits. Such databases are often implemented as a simple file bundled with the software, listing rootkits and the signatures specific to them. For example, the OSSEC host-based intrusion detection system includes a database of rootkits and specific files used or generated by them as a part of its detection toolset (OSSEC Project Team, 2019).

Behaviour-based detectors check for changes in system behaviour – such as the number of system calls executed by a specific program – that may be caused by rootkits (Todd et al., 2007). For example, a behaviour-based rootkit detection system using hardware performance counters (HPCs) found in modern CPUs has been designed to detect

hooking of various Windows kernel data structures. The system captures information from a number of HPCs – for example, the number of branches executed or caches prefetched – and attempts to detect changes in them. To establish a baseline for rootkits' effects on the HPCs' data, the system is bootstrapped by collecting HPC traces with synthesized rootkits (representing a specific type of rootkit behaviour) injected. The system uses machine learning models to classify the collected HPC traces as either clean or infected to train it to detect new rootkits exhibiting similar behaviour. (Singh, Evtyushkin, Elwell, Riley & Cervesato, 2017.)

Cross-view-based detectors gather system information (e.g. open sockets, running processes) from multiple interfaces and observe differences in their output to determine if one (or more) of them has been modified by potential rootkits (Todd et al., 2007). As an example of a cross-view-based detector, the deprecated RootkitRevealer tool for Windows compares the information from the Windows API with file system or registry contents (for example, a list of files) to check if they are consistent with each other, and alerts the user in case they're not (Russinovich, 2006b).

Integrity-based detectors gather snapshots of the system state (e.g. checksums of memory contents) and compare them against system state snapshots that are known to be valid and uncompromised (Todd et al., 2007). They are relatively common, with widely used tools such as Rootkit Hunter and Chkrootkit incorporating integrity checks as part of their detection mechanisms. For example, Rootkit Hunter uses a database of file properties (e.g. SHA-256 hash, inode number, file size, modification date, user and group IDs and permissions) gathered from a variety of system binaries, which it compares against when running a system check to confirm the properties have not been tampered with (Rootkit Hunter, 2018).

Finally, hardware-based detectors are devices which monitor changes made to kernel data structures and functions as well as search for signatures in memory that may signify a rootkit's presence (Todd et al., 2007). Such detectors are quite rare, although a few products do exist in the market. One example of a hardware-based detector, a kernel integrity monitoring system called Copilot, has been designed as a separate device used on the PCI bus. The system's authors claim it can read the physical memory of a running operating system using direct memory access (DMA) and continuously monitor the memory used by the Linux kernel using MD5 hashes to detect modifications to its critical data structures such as the system call table. (Petroni, Fraser, Molina & Arbaugh, 2004.)

In addition to categorization based on functionality, detectors can also be classified in terms of their execution context. Three approaches to rootkit and malware detection in virtual machines have been described: host- and network-based protection and virtual machine introspection. Host-based protection runs in the guest operating system itself, functioning as a host-based intrusion detection system (HIDS). Network-based protection, on the other hand, works at the hypervisor level, running intrusion detection on the inbound and outbound network traffic outside the guest operating system's reach. (Hua & Zhang, 2015.) Finally, virtual machine introspection (VMI) uses the virtual machine monitor (VMM) software to inspect the entire state of the virtual machine, so it has full visibility of the guest OS's behaviour while operating outside of it and thus remaining invisible to it (Garfinkel & Rosenblum, 2003).

Extensive research on developing rootkit detectors using advanced detection mechanisms has been conducted. A detector has been demonstrated that can operate outside of virtual machines using virtual machine introspection and reconstruct their

data to detect hidden processes with a higher degree of tamper resistance than detectors that run inside the guest OS (Hua and Zhang, 2015). A cross-view-based rootkit detector leveraging VMI has also been demonstrated. It uses VMI to obtain both the external view (memory images of a virtual machine instance) and the internal view (outputs of commands run inside the guest VM using APIs provided by the VMM) of a running VM and conducting view comparison to observe and detect differences – often indicative of potential system manipulation by rootkits – between them. (Richer, Neale & Osborne, 2015.)

An integrity-based detector targeting LKM rootkits using statistical methods has also been developed. The detector uses outlier analysis to compare the distribution of addresses stored in the kernel's system call table between the targeted system and a known uncompromised system. The authors of the detector state that the detector does not require existing knowledge of a particular system's state prior to an infection. The detector is successful in detecting rootkits with a "high degree of confidence", based on experiments conducted with two separate LKM-based rootkits. (Wampler & Graham, 2007.)

The detection tools used in this study can be categorized under signature-based, behaviour-based, cross-view-based and integrity-based detectors. Only hardware-based detectors are not covered out of the categories described by Todd et al. (2007), so the range of detection mechanisms should be sufficiently varied to establish a relatively broad coverage in the scope of this study. Also, the detectors are all host-based, running inside the guest operating system. This means that no network-based detection, virtual machine introspection or any other out-of-band detection mechanisms are used.

3. Methodology

This chapter describes the research methods, tools and processes used to conduct the empirical research in this study. The rootkits and rootkit detection tools selected for this study are also presented. Finally, the procedure used to evaluate the tools' effectiveness is described.

3.1 Research method

This study was an empirical research in which the effectiveness of rootkit detection tools' ability to discover the presence of various rootkits was evaluated and compared. Specifically, the study can be classified as a trial experiment where properties of a system are evaluated or tested to determine how well they meet their specifications or other criteria (Tedre, 2014). In the context of this study, the system consisted of the rootkit detection tools installed in the operating system and the evaluated property was the effectiveness of the detection mechanisms used by the tools to detect rootkits, which were collectively evaluated by injecting rootkits, running the detection tools individually and finally recording and analysing the results of the detection tests.

In addition to being a trial experiment, this study can also be classified as a comparison experiment, where a number of solutions are compared based on some criteria set to determine which of them are the best for a specific problem (Tedre, 2014). In this study, the solutions were the rootkit detection tools, the criterion was the tools' effectiveness in detecting rootkits (or indications of them) and the problem was rootkit detection in a Linux-based operating system.

An emulator-based experimental research technique was utilized in which real programs were executed inside an emulated environment. Two types of emulated environments or emulators can be specified: a sandboxed execution environment confined inside a virtual machine and a bare-metal execution environment where no virtual machine is used but the execution properties of the hardware are altered to emulate specific conditions. (Gustedt, Jeannot & Quinson, 2009.) This study opted for the former option, as real-world programs (rootkit detection tools and rootkits) were executed in a virtualized environment (a virtual machine) that closely emulates a real-world execution environment and context.

An emulation experiment is contrasted with three other types of experiments in terms of the application and the execution environment: in-situ experiments, simulations and benchmarks. In in-situ experiments, real-world programs are executed in a real-world environment without any emulation or simulation present. In simulations, a model of the program under investigation is executed in a simulation that abstracts away parts of the environment that are irrelevant for the study. In benchmarks, a model of the program is executed in a real-world environment with a focus on its quantitative instrumentation. (Gustedt, Jeannot & Quinson, 2009.)

There were two primary reasons for choosing an emulation-based experiment environment over the other environments. The first was repeatability; emulating the

environment using virtual machines allowed the detection tests to be easily and accurately repeated between test runs to double-check the test results, investigate potential issues and reduce overall variability of the result data. The second was security, as virtual machines provide some isolation between the guest and host operating systems by virtue of virtualization. As rootkits can be (and often are) used for malicious purposes, it made sense to take precautions to prevent potential attacks against the testing system.

3.2 Rootkits

A total of 15 rootkits were used in this study. The rootkits were split into two categories (user mode and kernel mode) for a representation of 6 user mode and 9 kernel mode rootkits. The included user mode rootkits were Azazel, Bedevil, BEURK, JynxKit2, Vlany and Zendar. The kernel mode rootkits were Diamorphine, Honey Pot Bears Rootkit, Keysniffer, LilyOfTheValley, Nuk3 Gh0st, Puszek, Reptile, Rootfoo Linux Rootkit and Sutekh.

Three selection criteria were applied for the rootkits in this study. First, they had to be open sourced and freely available online. Secondly, the kernel mode rootkits had to also support the kernel version used by the Linux distribution selected for testing. Finally, they had to adequately fit the rootkit definition criteria described by Harley and Lee (2007), which specifies rootkits as a type of software with the objective of enabling continuous, privileged and arbitrary control and access to a computer system while hiding or managing access to its contents such as files, processes and ports. While restricting the selection of rootkits to those that are publicly available does potentially limit the generalizability of the study, it was deemed too risky and cumbersome to attempt to collect “real-world” malicious rootkits in the wild using research honeypots or other collection mechanisms.

3.2.1 User mode rootkits

JynxKit2 is a user mode rootkit based on an older rootkit, JynxKit. It features rudimentary hiding of files and processes. The rootkit relies on the preloading mechanism used by Linux’s dynamic linker and loader (`ld-linux`) to override functions defined in other libraries to hook into programs that are to be infected. The hooks are used to intercept calls to shared objects and system calls in an attempt to hide the rootkit’s presence in the system. It features a backdoor based on the `accept()` system call, which can be used to drop a reverse shell in the target system to allow remote access and operation. (ErrProne, 2011b.)

Azazel is a user mode rootkit based on JynxKit2, with features such as file, login, connection and process hiding, utmp and wtmp log manipulation as well as backdoors. It also uses preloading for injection. The rootkit can hide files and directories in the file system whose names include a string configured at compilation time. It also includes features to avoid detection and debugging, such as string obfuscation and hooking the PCAP packet capture API (to hide backdoor traffic) and `ptrace()`. The rootkit includes two backdoors using `accept()` and another encrypted backdoor using PAM, which can also be used to grant root privileges to users connecting through it. The ports used by the backdoors and other remote connections are hidden by the rootkit. (Chokepoint, 2017.)

BEURK Experimental Unix Rootkit is an another user mode rootkit with a similar set of features as in Azazel, such as file, process and login hiding, `accept()` backdoors and anti-detection/debugging countermeasures. The rootkit can be used to hide files and directories whenever their names contain a predefined string. It can also obfuscate strings and automatically remove traces of its actions from utmp and wtmp logs. Like other user mode rootkits, it too relies on preloading to inject itself to programs. (Unix-Thrust, 2017.)

Vlany is another `LD_PRELOAD` -based user mode rootkit with a variety of anti-detection/debugging features. It can hide files, processes, users, network connections and includes a PAM backdoor for SSH access in addition to the `accept()` backdoor in the other rootkits described above. It also features mechanisms to prevent its removal, including modifying the operating system's dynamic linker. The rootkit hides processes in the system whenever the process owner's group ID (GID) matches a particular GID, specified at compilation time. The author of the rootkit claims that the process hiding feature is more advanced and dependable than Azazel's hiding mechanisms and that the rootkit's PAM backdoor is an improvement over the `accept()` backdoors. (Mempodippy, 2019.)

Bedevil is a rootkit by the author of Vlany, created as an effort to improve upon its design and implementation. It features file and process hiding based on GIDs, logging of user authentications and outgoing SSH credentials as well as a PAM backdoor with encryption and utmp/wtmp hooking (similar to the one found in Vlany). It has rudimentary methods for detecting when it's being run in virtualized environments such as VirtualBox (which it detects by searching for the string `VBOX` in `/proc/scsi/scsi`). It can also hide its presence in process memory region mapping information in the Linux `/proc` file system. (Rowan, 2019.)

Zendar is a simple user mode rootkit utilizing the preloading mechanism for injection. It disguises itself as a shared library (`libsslcore.so`) and hides the file by default to prevent detection by cursory inspection and detection tools. The rootkit can also hide files with names matching a specific string. For backdoor access, the rootkit creates a user who can override hooking and obfuscation and whose existence is also hidden from the `/etc/passwd` and `/etc/shadow` files. (Ring-1, 2015.)

Table 2 compares the features of each user mode rootkit used in this study. The column named self indicates that the rootkit can conceal itself, file indicates that the rootkit can hide files or directories, process indicates that the rootkit can hide tasks or processes, and connection means that rootkit is able to hide sockets, ports or other indications of network connections. Logging indicates that the rootkit can capture or log data such as entered keystrokes or login credentials. Cells marked with an x indicate that the feature is supported by the rootkit, while an empty cell means that the feature is not supported by the rootkit.

Table 2. Comparison of user mode rootkit features.

Rootkit	Self	File	Process	Connection	Logging	Backdoor
JynxKit2	x	x		x		x
Azazel	x	x	x	x		x
BEURK	x	x	x	x		x
Vlany	x	x	x	x		x
Bedevil	x	x	x		x	x
Zendar	x	x				x

All user mode rootkits selected for this study can hide themselves as well as files and directories. Most rootkits can also hide processes and connections. Only Bedevil contains logging features. All user mode rootkits also contain a backdoor feature to enable remote access.

3.2.2 Kernel mode rootkits

Diamorphine is a kernel mode rootkit that can hide itself as well as any processes, kernel modules and files its operator chooses. It is implemented as an LKM, allowing it to intercept system calls and override the kernel's functionalities, giving it practically full access to the system. At compilation time, a magic string can be specified for the rootkit which is then used to hide files or directories whose names are prefixed with it. To hide/unhide the rootkit and grant a user root privileges, specific signals can be sent to any process to trigger the mechanisms. Hiding or unhiding a specific process works by sending that process a predefined signal. The rootkit does not include a backdoor, so it needs to be combined with other software to enable remote access and operation. (Mello, 2019.)

Honey Pot Bears Rootkit is another kernel mode rootkit implemented as an LKM. It can give root privileges to processes that receive a predetermined signal by changing their user ID (UID) to 0. It can also hide files with names matching a specific prefix and processes with a specific name or process ID (PID) configured by the attacker. The rootkit can also create a user account hidden from anyone inspecting the `/etc/passwd` and `/etc/shadow` files to facilitate elementary backdoor access. (Kleiman, Gao, Khan & Song, 2019.)

Keysniffer is an LKM that can be classified as a kernel mode rootkit due to its kernel hooking capabilities. It is a keylogger that records keyboard events to debugfs, an in-memory file system used for Linux kernel debugging. It does not feature any particular anti-detection mechanisms except that it prevents accessing the log file by non-root users and the module is named "kisni.ko" by default to avoid immediate suspicion when listing kernel modules. (Jana, 2019.) While Keysniffer may not meet the criteria of all rootkit definitions due to its lack of specific hiding functionalities, it does meet the criteria by Harley and Lee (2007) and is thus included in this study.

LilyOfTheValley is an LKM rootkit with file and process hiding as well as privilege escalating features. It can hide itself as well as files with a specific prefix in their names. It can also give root privileges to users interacting with the kernel module through the

`/proc` file system. The rootkit also comes with a client program that can be used to hide and unhide arbitrary processes in the system by any user who has permissions to execute the program. (Algayar, 2017.)

Nuk3 Gh0st is an another LKM-based rootkit with multiple different hiding features. It is able to hide itself from kernel module lists to prevent detection by basic manual inspection. The rootkit can also hide arbitrary files and directories in the file system as well as any processes and TCP/UDP ports. It can also give root privileges to users and prevent the module's removal by kernel module manipulation commands such as `rmmod`. The rootkit comes with a separate client program which can be used to control all of the aforementioned functions. (Schällibaum, 2018.)

Puszek is an LKM-based kernel mode rootkit which allows files and processes (including their connections) to be hidden in the system. It can hide files and directories in the file system whenever their names match a predefined suffix. It can also hide processes where any part of a process's command line matches a predetermined string. It is also capable of capturing and logging HTTP requests and can detect and store passwords found in the requests to a separate log file. The rootkit can use two separate methods of modifying the system call table's write permissions for improved compatibility, either by manipulating the CR0 control register directly or using a kernel helper method to set the system call table's page writable. In addition, the rootkit can hide itself from the `/proc` file system along with other module listings and can also prevent attempts to remove it. (Eterna1, 2018.)

Reptile is an another LKM-based x86-64 kernel mode rootkit with features such as arbitrary privilege escalation as well as file, process and connection hiding. In addition to hiding files in the file system, the rootkit can hide content within files, whenever the content is enclosed within special tags (`<reptile></reptile>` by default). It can also persist itself between reboots and includes anti-detection functionalities such as string obfuscation and file encryption. The rootkit includes an encrypted backdoor which can spawn a reverse shell upon receival of a magic packet from a separate client program that is included as a part of the software package. (Augusto, 2019.)

Rootfoo Linux Rootkit is an LKM rootkit developed as a demonstration of a kernel mode rootkit and its injection and manipulation capabilities. The rootkit can execute a program installed in a specific location inside a kernel thread to elevate the privileges of the process. It can also hide itself from the `/proc` file system and can prevent detection by basic manual inspection as well as module listing tools such as `lsmod`. (Hodges, 2019.)

Sutekh is an LKM-based kernel rootkit with the primary intention of granting root privileges to user mode processes. It works by hooking the `execve()` and `umask()` system calls in a copied version of the system call table and comes with a separate privilege escalation program that can give any user executing it immediate root privileges by spawning a root shell. The rootkit does not feature any particular evasion or anti-detection mechanisms. (PinkP4nther, 2019.) Like Keysniffer, Sutekh doesn't meet the all of the criteria of more strict rootkit definitions, though it does fit the definition by Harley and Lee (2007).

Table 3 compares the features of each kernel mode rootkit used in this study. The first four columns after the rootkit name are related to the rootkits' hiding mechanisms. Self indicates that the rootkit can conceal itself, file indicates that the rootkit can hide files or directories, process indicates that tasks or processes can be hidden by the rootkit, and

connection indicates that the rootkit can hide sockets, ports or other indications of network connections. The privileges column indicates that the rootkit supports elevating the privileges of a process to root, while the logging column indicates that the rootkit can capture or log system events such as keystrokes entered by users. Cells marked with an **x** indicate that the rootkit supports the feature, while an empty cell indicates that the feature is not supported.

Table 3. Comparison of kernel mode rootkit features.

Rootkit	Self	File	Process	Connection	Privileges	Logging	Backdoor
Diamorphine	x	x	x		x		
Honey Pot Bears		x	x		x		x
Keysniffer						x	
LilyOfTheValley	x	x	x		x		
Nuk3 Gh0st	x	x	x	x	x		
Puszek	x	x	x	x			
Reptile	x	x	x	x	x		x
Rootfoo Linux Rootkit	x				x		
Sutekh					x		

Most kernel mode rootkits selected for this study can hide themselves as well as files, directories and processes. Most can also provide privilege escalation for arbitrary processes. Only three can hide connections, one provides logging features and two feature a backdoor option.

3.3 Rootkit detection tools

Five software tools with capabilities suitable for detecting rootkits were chosen for this study: OSSEC, AIDE, Rootkit Hunter, Chkrootkit and LKRG. Each tool is available as open source software, supports the Linux operating system and is actively supported (updated more than once in 2019). Ensuring that the detection tools were relevant was considered an important factor in upholding the relevance of this study itself, as the results could be more applicable to real-world scenarios where such tools are used for rootkit detection purposes.

Other detection tools studied and considered for this research were Samhain (Wichmann, 2006), Open Source Tripwire (Tripwire, 2019), ClamAV (Cisco, 2019), Tiger (Schales, Hess, Warraich & Safford, 2019) and Linux Malware Detect (R-fx Networks, 2019). However, these tools were not chosen due as they were not considered relevant or feasible enough for rootkit detection in this study.

Samhain was not chosen because of its complex configuration and extremely verbose output which would have made interpreting its results too time consuming and complicated. Open Source Tripwire was not chosen because of its overlapping feature set with AIDE, which was deemed sufficient for covering file integrity detection in the scope of this study. Tiger was not chosen because it uses an already included tool (Chkrootkit) for rootkit detection. ClamAV and Linux Malware Detect were not chosen because they do not include facilities for rootkit detection but instead focus on signature-based detection of viruses and worms.

OSSEC is an open source host-based intrusion detection system for multiple platforms, including Linux, BSD and Windows. It provides an agent-based continuous system monitoring solution with features such as log monitoring, integrity checking and rootkit detection. Rootkit detection in OSSEC includes mechanisms such as detecting binary modifications, checking for hidden processes and scanning for promiscuous network interfaces. The mechanisms are a part of a "rootkit detection engine" named Rootcheck, which is included in the core product. (OSSEC Project Team, 2019.)

Based on the categories by Todd et al. (2007), OSSEC can be categorized as a signature-based detector as it checks files created by known rootkits against a signature database. It can also be categorized as a behaviour-based detector as it inspects files and directories for deviation from expected configurations. The tool can also be considered a cross-view-based detector as it checks for the existence of e.g. hidden processes by comparing outputs of different commands and observing differences between them. In addition, OSSEC can be categorized as an integrity-based detector as it can detect modifications made to the file system.

AIDE – short for Advanced Intrusion Detection Environment – is an open source file integrity checker. It provides file integrity verification by monitoring an operating system's file system for modifications. The tool stores properties of the file system's contents, such as file attributes and checksums in a database file and detects if they have changed between scans to inform about (possibly malicious and unauthorized) manipulation of the file system's contents. The properties to store in the AIDE database can be specified by the user for any configuration of file paths based on regular expressions and sets of file properties to collect for each matching path. (Lehti & Virolainen, 2019.)

AIDE can be categorized as an integrity-based detector as it monitors the file system for modifications. While not specifically designed for rootkit detection, AIDE can be used to identify changes made in the file system by rootkits and is in fact one of the tools suggested by the authors of Rootkit Hunter to complement the functionality of rootkit detectors (Rootkit Hunter, 2019).

Rootkit Hunter – also known as rkhunter – is an open source rootkit detector for Unix-based operating systems. It is implemented as a Unix shell script. The tool attempts to detect rootkits by checking file properties, looking for suspicious software configurations and abnormalities in kernel modules and system binaries among other mechanisms. As its name suggests, its primary focus is on rootkit detection, though it may detect traces of other malware as well. The tool can also be configured to use Tripwire for additional verification of the file system's integrity, as well another tool called Unhide for additional checking of hidden processes and ports. (Rootkit Hunter, 2018.) However, the optional tool integrations are not used in this study to keep the detector configuration as "normal" as possible.

Rootkit Hunter can be categorized as a signature-based detector as it contains signatures for specific rootkits. It can also be considered a behaviour-based detector due to its capability for checking system configurations. The tool can also be considered an integrity-based detector as it is able to collect and compare file properties between executions.

Chkrootkit is another open source rootkit detection tool for Unix-based operating systems. It is similar to Rootkit Hunter in that it is also implemented as a Unix shell script with comparable features such as checking for hidden files, system binary

modifications, rootkit signatures and system configurations. It can also check for the modification of logfiles in the system. The tool can be used to detect multiple worms and trojans as well as other types of malware, though its main focus is on detecting rootkits. (Murilo & Steding-Jessen, 2019.)

Chkrootkit can be categorized as a signature-based detector as it includes signatures for known rootkits. It can also be categorized as a behaviour-based detector as it can check for unusual configurations of system binaries. Also, it can be considered a cross-view-based detector as it can check for e.g. inconsistencies between process properties in the `/proc` file system and related system calls.

LKRG (Linux Kernel Runtime Guard) is a kernel integrity checker and exploit detector specifically designed for the Linux kernel. It can monitor the state of key Linux kernel data structures and detect modifications by storing and comparing addresses, checksums and properties of data structures such as kernel module object lists, interrupt descriptor and exception tables as well as control registers. It can also detect active kernel exploitation by checking the integrity of process IDs, sandboxing states and SELinux variables among other attributes. LKRG can also hide itself from kernel module listings to remain stealthy and to counter some anti-evasion features employed by malware. (Openwall, 2019.)

LKRG can be categorized as a behaviour-based detector as it checks for specific values in key pointers and variables of the kernel's data structures. Because it can check and compare data (e.g. kernel module listings) from multiple sources, LKRG can also be categorized as a cross-view-based detector. As LKRG's main features are strongly related to checksum and property-based comparisons, it can be also categorized as an integrity-based detector.

Table 4. Rootkit detection tool categories.

Rootkit	Signature	Behaviour	Cross-view	Integrity
OSSEC	x	x	x	x
AIDE				x
Rootkit Hunter	x	x		x
Chkrootkit	x	x	x	
LKRG		x	x	x

Table 4 shows the categories each rootkit detection tool used in this study are included in. The categories are based on rootkit detector classification described by Todd et al. (2007). All of the detector categories except hardware-based detectors are represented in this study to provide a wide-ranging coverage of the detection mechanisms used by the tools.

3.4 Test environment

The environment in which the rootkit detectors were tested consisted of a Linux distribution running as a guest operating system in a virtualized environment. The virtual machine monitor used to manage the guest operating system was Oracle VM VirtualBox version 5.2.22, a hosted type 2 hypervisor that supports full virtualization with hardware-assisted virtualization (Oracle, 2020).

As discussed earlier, there were two primary reasons for using a virtual machine based emulated testing environment in this study: repeatability and security. The hypervisor allows the state of the virtual machine to be saved as a "snapshot" at any given moment, to which the virtual machine could then be reverted to later on. This made it possible to define the common initial state for each test run, preventing modifications to the environment made by the rootkits and/or detectors from persisting and affecting other test runs.

Another benefit of virtualization is that it also provides a layer of isolation between the host operating system and the software running inside the virtual machine. This helps decrease the likelihood of malicious code gaining access beyond the guest operating system. (Hyde, 2009.) However, the guest operating system can still communicate with the host OS over the network, presenting a potential attack vector for the rootkits to attempt to escape containment.

In the test environment, outbound network communications from the guest operating system were restricted by using an application-level firewall to only allow outbound traffic to ports 53 (DNS), 80 (HTTP) and 443 (HTTPS) while blocking unestablished inbound traffic (except for port 22 in the rootkit installation stage to allow copying files over SSH). Also, outbound communication to the host operating system and other devices in the LAN (excluding the router) was blocked by firewall rules to prevent the guest operating from initiating connections to local devices on its own for potentially malicious purposes.

As mentioned earlier, the source code of each rootkit was also inspected and cursorily audited to help prevent potential security breaches. No rootkits were found to contain execution paths which would attempt to subvert the system beyond what was described in their documentation. These considerations allowed the rootkits to be tested with increased confidence that they wouldn't be able to escape the virtual machine and cause external damage, though this can never be completely guaranteed.

Ubuntu 16.04.6 LTS (Xenial Xerus) was used as the testing environment for executing test runs. The distribution is well established and used for desktop, server and other applications. At the time of conducting this research, the distribution was still receiving continuous maintenance and security updates. The Linux kernel version used by the distribution in this research was 4.15.0 and the instruction set architecture was x86-64. The full system information string for the distribution using the `uname -a` command was `Linux ubuntu1604 4.15.0-45-generic #48~16.04.1-Ubuntu SMP Tue Jan 29 18:03:48 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux`.

The Ubuntu 16.04 distribution version was chosen over more modern versions such as Ubuntu 18.04 due to its kernel version being the most compatible with the kernel mode rootkits used in this study. Ubuntu 18.04 (and later versions) uses the much newer 5.x kernel series, which most of the kernel mode rootkits surveyed in the initial stages of this study did not support. However, it should be noted that many kernel mode rootkits evaluated for inclusion in this study did not support kernel version 4.15.0 used by Ubuntu 16.04 either, which posed a challenge by limiting the number of usable kernel mode rootkits to test.

3.5 Test process

The testing process consisted of three stages. It was partly inspired by a similar testing method described by Richer, Neale and Osborne (2015), where rootkits were injected to a common initial state for each test using a virtual machine environment with the state of the system in specific points of time stored as snapshots, facilitated by the virtual machine monitor. As opposed to that study, this one did not use any third-party tools for injecting the rootkits or recording data about them, instead relying on simple custom shell scripts for automating the injection and data extraction procedures.

In the first stage, a virtual machine image was created using VirtualBox with the following resource specifications:

- Operating system: Linux (Ubuntu 16.04.6) x86_64
- CPU: 4 virtual CPUs
- Main memory: 4096 MB
- Video memory: 128 MB
- Disk: 10 GB dynamic VDI

Port forwarding was set to open TCP port 22 for the guest operating system for SSH access from the host operating system to copy files into the virtual machine. The Ubuntu distribution was then installed, and the system's packages were updated to the latest available versions. A "clean" snapshot was created from this state, serving as the common initial state that was used for each rootkit detector configuration.

In the second stage, each of the rootkit detectors used was installed and configured in their separate virtual machines. A snapshot was created for each of the detectors in a state where they were immediately ready to be used. The snapshots were used as the starting point for each individual detection test. As discussed in the description of the test environment, this helped ensure that the tests could be easily repeated as many times as needed with minimal variability.

Each of the stage two snapshots were loaded with two scripts: a launcher script and a logger script. The launcher script's purpose was to execute a rootkit detection tool with a predetermined configuration to conduct a full detection run. The logger script then captured and moved the detector's output to a specified location for retrieval and ensured it was ready to be accessed from the host operating system. For detailed installation, configuration and execution steps, see Appendix B.

In the third stage, a rootkit was injected to each of the stage two virtual machine snapshots using detailed instructions derived from trial runs (see Appendix C). Once the rootkit was injected, a snapshot was taken to ensure the injection could be verified without modifying the system in an actual detection test. The detector installed in a snapshot was then run using the launcher script. Once the detector had finished running, the logger script was executed to record and copy the results of the detection run for further analysis.

The test runs were executed serially, with only one virtual machine instance running at a time to ensure that performance remained stable between test runs and that no other forms of interference were introduced. In addition to detection runs with a rootkit injected, a clean run was also conducted where no rootkit had been installed to capture a control result to compare against the other detection runs.

While a “real-world” rootkit injection scenario would very likely include a chain of exploits including a dropper program, the test process in this study did not use any such tools in the injection process. This was to eliminate extraneous variables that could affect the detection results as much as possible. In addition, ensuring that no additional exploits or tools were introduced to the system made the detection process a “worst-case”-scenario for the rootkit detection tools, simulating a scenario where any extra traces of infection had been removed, thus levelling the field for each test run.

After all detection runs were completed, i.e. each detector had finished running with each rootkit installed on each distribution, the obtained results were used for analysis. The maximum duration specified for a single injection-detection run was 30 minutes, serving as a cut-off mechanism to ensure that all tests could be executed in a reasonable amount of time.

3.6 Evaluation procedure

The effectiveness of each detection tool was evaluated to determine how successful the tool was in finding a specific rootkit and its corresponding rootkit category (user mode and kernel mode). The evaluation considered five options for each detection run: whether the detector was able to explicitly detect a rootkit, provide warnings about a potential infection, unable to successfully finish the test without exhibiting abnormal behaviour, provide a false positive result unrelated to rootkit infection or miss the rootkit’s existence entirely.

The options were categorized into two kinds of detections: positive and negative detection indications. The positive detection indication category combined explicit detections, potential infection warnings and detection results indicating changes in system behaviour that could be caused by rootkits, malware or other system exploitation. The negative detection indication category combined abnormal test executions without warnings, false positives and detection results that were identical to the clean runs. The categorization was performed to enable a more straightforward and conclusive comparison between the detection tools.

4. Results

This chapter presents the results of the rootkit detection tests conducted in this study. In addition, the phases of the detection test runs are described for each detection tool. Finally, the results of the detection tests are summarized.

4.1 Detection runs

Each detector was installed and configured from a "clean" virtual machine memory snapshot, where the Ubuntu distribution was installed and configured to represent a fresh operating system installation. Once configured, another snapshot was taken for each detection tool as a starting point for the detection runs.

Each detection run was executed by injecting a rootkit to the system, capturing a VM snapshot directly after injection, executing the rootkit detection tool and capturing the test's results. After each run, the state of the VM was restored using the snapshot where the detector had been configured. The rootkit injection was verified separately for each rootkit by restoring the snapshot where the rootkit had been injected. For full installation, configuration and detection run procedures for each detection tool, see Appendix B. For detailed rootkit installation and configuration procedures (including injection verification), see Appendix C.

A total of 75 detection tests were conducted, excluding clean, trial and confirmation test runs. Each detection tool was tested against each rootkit, so no combination of detector and rootkit was skipped. In some cases, detection runs were repeated to confirm that the results were not affected by external events.

4.2 OSSEC

In preparation for detection tests, OSSEC version 3.5.0 was downloaded from its official GitHub code repository and installed using the included install script. OSSEC's Syscheck and Rootcheck checks were determined to likely exceed the 30-minute cut-off based on initial test runs. Therefore, they were configured to execute faster by removing a time delay that would have normally been triggered after scanning a certain number of files or ports.

In the clean run, OSSEC's Rootcheck detected that the file `/run/user/1000/gvfs` was related to a "possible kernel level rootkit" due to a cross-view mismatch where the file was visible for `readdir()` but not for the `stat()` system call. This was determined to be normal behaviour for the system and therefore a false positive. The result output displaying the false positive is shown in Figure 4.

```
...
2020 Jan 29 19:01:23 (first time detected: 2020 Jan 29 19:01:23)

System Audit: Anomaly detected in file '/run/user/1000/gvfs'. Hidden
from stats, but showing up on readdir. Possible kernel level rootkit.
```

Figure 4. OSSEC clean run false positive.

OSSEC was able to provide warnings of the existence of a potential rootkit in the system for JynxKit2, Zendar, Puszek and Reptile. In the detection run for JynxKit2, OSSEC detected link count mismatches in directories `/`, `/etc`, `/var/spool`, `/var/cache` and `/var/log/hp`, `readdir()/stat()` mismatches in files `/etc/cups`, `/run/cups`, `/var/cache/cups`, `/var/spool/cups` and `/var/log/hp/tmp` as well as a single process hidden from `ps`. All of the results were consistent with the behaviour of JynxKit2 based on an inspection of its source code and reference documentation.

In the case of Zendar, the files `/lib/libsslcore.so` and `/etc/ld.so.preload` (both manipulated by Zendar) were detected as anomalous by OSSEC with a cross-view mismatch between `readdir()` and `stat()` system call results. In addition, the existence of a non-root user account with UID 0 created by Zendar was detected in `/etc/passwd`. The results were consistent with Zendar's code and reference documentation.

In the detection run with Puszek injected, OSSEC detected a cross-view mismatch in the file `/etc/^A`. Puszek creates hidden files `/etc/modules`, `/etc/http_requests` and `/etc/passwords` for logging and hiding purposes, so the cross-view mismatch was very likely connected to them. The detection run was re-executed twice to rule out other causes, and their results were identical to the first run.

Figure 5 shows the most relevant portions of the OSSEC rootcheck results when attempting to detect Reptile. OSSEC warned about a hidden file in the root directory, triggered by its cross-view-based detection checks. The file was determined to be `/reptile/reptile_cmd` based on a manual inspection conducted after the detection run. Also, a mismatch in the file size of `/etc/modules` and the link count of `/lib/modules/4.15.0-45-generic/kernel/drivers/PulseAudio` was detected by OSSEC and classified as a potential rootkit. Both files are manipulated by Reptile, confirming that the results were valid.

```
2020 Jan 29 19:06:06 (first time detected: 2020 Jan 29 19:06:06)
System Audit: Files hidden inside directory '/lib/modules/4.15.0-45-
generic/kernel/drivers/PulseAudio'. Link count does not match number
of files (2,3).

2020 Jan 29 19:06:06 (first time detected: 2020 Jan 29 19:06:06)
System Audit: Anomaly detected in file '/etc/modules'. File size
doesn't match what we found. Possible kernel level rootkit.

2020 Jan 29 19:06:09 (first time detected: 2020 Jan 29 19:06:09)
System Audit: Files hidden inside directory '/'. Link count does not
match number of files (24,25).
```

Figure 5. OSSEC results summary for the Reptile detection run.

Besides direct indications of rootkits JynxKit2, Zendar, Puszek and Reptile, OSSEC found suspicious system behaviour in the detection runs for Bedevil and Vlany. Detection run results for Azazel, BEURK, Diamorphine, Honey Pot Bears, Keysniffer, LilyOfTheValley, Nuk3 Gh0st, Puszek, Rootfoo Linux Rootkit and Sutekh were identical to the clean run, with no indications of rootkit injection or suspicious behaviour given by the tool.

In Bedevil's detection run, OSSEC detected a mismatch in the number of links to directory `/usr/share/doc/libcolord2`. The directory was created by Bedevil during its injection phase to host the rootkit executable and its configuration. This means that the warning given by OSSEC was relevant.

In Vlany's detection run, OSSEC detected a link count mismatch in the `/lib` directory, which is where Vlany installs its shared object file for `LD_PRELOAD` hooking. However, it was unable to detect any specific files manipulated by Vlany, making actual identification of the rootkit difficult.

4.3 AIDE

To prepare detection tests with AIDE, the latest released version (0.16.2a2-19-g16ed855) was first installed on the system using Ubuntu's APT package manager. Then, its database of file and directory properties was initialized with `aideinit`. After database initialization, an AIDE configuration file was generated for the system and set as AIDE's default configuration.

Since AIDE doesn't explicitly detect the existence of rootkits, interpreting its results required additional analysis of the rootkits' behaviour. Specifically, this meant that the files and directories that were created, deleted or modified by rootkits needed to be determined based on available references as well as rootkit source code and then compared against AIDE's results to find matches.

A summary of results for an AIDE integrity check run against the clean system without rootkits installed is shown in Figure 6. Constant changes in the file system were detected over the course of the AIDE test runs, mostly originating from the `/dev` and `/run` directories containing volatile device and runtimes files and directories.

```
Summary:
  Total number of entries:      235723
  Added entries:                7
  Removed entries:              5
  Changed entries:              11
```

Figure 6. AIDE results summary for the clean run.

Based on comparison of AIDE's results and the files/directories determined to be modified by the rootkits used in the detection tests, AIDE was able to detect modifications to files and directories made by eight of the 15 rootkits. All user mode rootkits left traces that were captured by AIDE, while only two of the nine kernel mode rootkits (Honey Pot Bears Rootkit and Reptile) left traces visible to AIDE in the file system.

For Azazel, AIDE detected the addition of files `/etc/ld.so.preload` and `/lib/libselinux.so` as well as modification of the `/lib` directory. However, it did not detect the addition of files hidden by Azazel using its magic string. In the Bedevil detection run, AIDE detected the addition of files `/etc/ld.so.preload`, `/etc/system/system/sysinit.target.wants/.17174`, multiple files added to `/usr/share/doc/libcolord2/.19465` as well as the modification of file `/etc/ssh/sshd_config`. All of the files added or modified by Bedevil were found.

AIDE found files `/etc/ld.so.preload` and `/lib/libselinux.so` used for injection by BEURK. It also detected files hidden by BEURK's magic string feature. It also found `/etc/ld.so.preload`, `/XxJynx/jynx2.so` and `/XxJynx/reality.so` created by JynxKit2, clearly showing the rootkit's injection. AIDE also detected every file/directory added or modified by Vlany, such as `/etc/ld.so.preload`, `/etc/ld.so.conf.d`, `/lib/ld-linux.so.2`, `/lib/libc.so.test.03`, `/opt/.54xJE6BG` and `/etc/ssh/sshd_config`. Vlany's injection left a rather large injection footprint on the file system with many suspicious files added or modified, so the existence of a rootkit was easy to discover in AIDE's results.

Most relevant parts of AIDE results summary for the Zendar detection run are shown in Figure 7. The added entries are especially suspicious and indicative of a user mode rootkit; the preload hooking, disguised shared object as well as rootkit user configuration and environment are clearly visible. The files were all confirmed to have been manipulated by Zendar for injection and persistence purposes.

```
Summary:
  Total number of entries:      235733
  Added entries:                16
  Removed entries:              8
  Changed entries:              23
---------------------------------------------------
Added entries:
---------------------------------------------------
...
f++++++++++++++++: /etc/.passwd
f++++++++++++++++: /etc/.shadow
f++++++++++++++++: /etc/ld.so.conf.d/.bashrc
f++++++++++++++++: /etc/ld.so.conf.d/login
f++++++++++++++++: /etc/ld.so.preload
f++++++++++++++++: /lib/libsslcore.so
...
---------------------------------------------------
Changed entries:
---------------------------------------------------
...
d =.... mc.. .. .: /etc/ld.so.conf.d
f >.... mc..C.. .: /etc/passwd
f >.... mc..C.. .: /etc/shadow
d =.... mc.. .. .: /lib
...
```

Figure 7. AIDE results summary for the Zendar detection run.

In terms of kernel mode rootkits, AIDE did not detect modifications to the file system made by Diamorphine, Keysniffer, LilyOfTheValley, Nuk3 Gh0st, Puszek, Rootfoo Linux Rootkit or Sutekh. It did, however, detect changes to files `/etc/group`, `/etc/passwd`, `/etc/shadow`, `/etc/subgid`, `/etc/subuid`, all of which are modified by the Honey Pot Bears rootkit for backdoor user installation. AIDE also detected the deletion of files including the string `secret`, which is the default magic string for hiding arbitrary files by the rootkit. The apparent deletion of such files can be quite clearly considered a red flag when analysing results, although the deletion also meant that AIDE was unable to track the existence of any files matching the rootkit's magic string.

AIDE was able to detect the addition of file `/lib/modules/4.15.0-45-generic/kernel/drivers/PulseAudio` injected by Reptile as well as modifications to files in `/lib/modules/4.15.0-45-generic` and `/etc/modules`. However, it did not detect the addition of crucial control and persistence files injected by Reptile to `/reptile`. It could be argued that the files detected by AIDE can be difficult to link to a rootkit injection, though doing so is dependent on how aware the person analysing AIDE's results is of the specifics of Reptile's injection behaviour.

4.4 Rootkit Hunter

In preparation for detection tests, Rootkit Hunter version 1.4.6 was installed using the APT package manager. During installation, the tool was configured not to send emails of detection results. After the tool was installed, the file property database used by

Rootkit Hunter's file integrity checker mechanism was initialized by executing the command `rkhunter --propupd`.

The clean Rootkit Hunter test run without rootkits installed produced some warnings; the commands `adduser`, `ldd`, `lwp-request`, `egrep`, `fgrep`, and `which` were determined to have been "replaced by a script", processes for `compiz` and `nautilus` had "suspiciously" large shared memory segments, SSH root login was found to be enabled and eight "suspicious file types" were detected in `/dev/`. In its summary, Rootkit Hunter reported six suspicious files and two potential rootkits. However, these warnings were deemed to be default behaviour for the distribution. The check summary report for the clean run is shown in Figure 8.

```
[21:47:09] System checks summary
[21:47:09] ====================
[21:47:09]
[21:47:09] File properties checks...
[21:47:09] Files checked: 147
[21:47:09] Suspect files: 6
[21:47:09]
[21:47:09] Rootkit checks...
[21:47:09] Rootkits checked : 501
[21:47:09] Possible rootkits: 2
[21:47:09]
[21:47:09] Applications checks...
[21:47:09] All checks skipped
[21:47:09]
[21:47:09] The system checks took: 1 minute and 14 seconds
```

Figure 8. Rootkit Hunter check results summary for a clean detection run.

Rootkit Hunter was able to directly detect four of the 15 rootkits tested: JynxKit2, Zendar, Diamorphine and Rootfoo Linux Rootkit. However, Zendar and Rootfoo Linux Rootkit were incorrectly detected as SucKIT and Diamorphine, respectively. Figure 9 shows the most relevant results of Rootkit Hunter checks when attempting to detect JynxKit2.

```
[23:07:45] Checking for Jynx2 Rootkit...
[23:07:45]   Checking for file '/XxJynx/reality.so'          [ Found ]
[23:07:45]   Checking for directory '/XxJynx'                [ Found ]
[23:07:45] Warning: Jynx2 Rootkit                            [Warning]
[23:07:45]          File '/XxJynx/reality.so' found
[23:07:45]          Directory '/XxJynx' found
...
[23:08:54] System checks summary
[23:08:54] ====================
[23:08:54]
[23:08:54] File properties checks...
[23:08:54] Files checked: 147
[23:08:54] Suspect files: 6
[23:08:54]
[23:08:54] Rootkit checks...
[23:08:54] Rootkits checked : 501
[23:08:54] Possible rootkits: 3
[23:08:54] Rootkit names    : Jynx2 Rootkit
[23:08:55]
[23:08:55] Applications checks...
[23:08:55] All checks skipped
[23:08:55]
[23:08:55] The system checks took: 2 minutes and 10 seconds
```

Figure 9. Rootkit Hunter check results for the JynxKit2 detection run.

The detection run for Azazel produced some abnormal detection results. As the test was executed, each line printed to the screen by Rootkit Hunter produced a display error. The errors were not visible in the logfile produced by Rootkit Hunter. This was because the actual result (e.g. OK, Warning, Not Found) for each check was simply not saved to the file. A screenshot displaying some of the errors is shown in Figure 10.

Also, according to the logs, system command checks were not run but instead skipped entirely without notification. Such errors could be perceived as a bug in the detector itself, though they may raise suspicions in some scenarios – especially as the clean runs produced correctly formed results output. The tool itself did not provide any indications that the errors could be related to malicious activity.

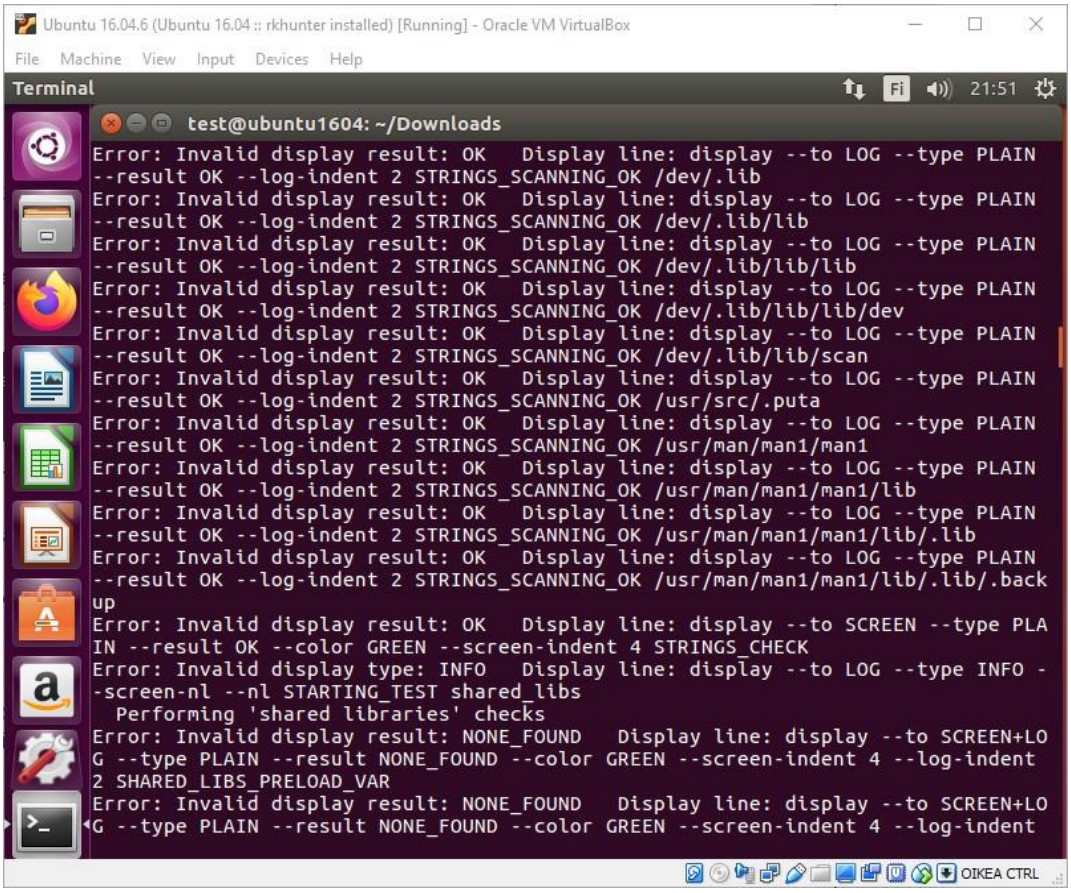

Figure 10. Rootkit Hunter display errors in the Azazel detection run.

In terms of user mode rootkits, Rootkit Hunter was unable to detect any traces of Bedevil, BEURK or Vlany. Of the kernel mode rootkits, Honey Pot Bears Rootkit, LilyOfTheValley, Nuk3 Gh0st, Keysniffer, Puszek, Reptile and Sutekh were also not detected at all. In these cases, the Rootkit Hunter results output was identical to the clean run results.

Interestingly, it appears that all of the successful detections were achieved by the rootkit signature checking mechanisms of Rootkit Hunter. This was especially apparent in the explicit detections. No indications of the behaviour or integrity checks contributing to the findings were discovered.

4.5 Chkrootkit

Chkrootkit version 0.52 was installed using the APT package manager. No further configuration was required, so Chkrootkit was immediately ready to be used for the detection tests. Chkrootkit doesn't save the results of a detection run to any logfile by default, so its output was redirected to both a logfile and standard output using the `tee` command. The command used to execute each of the Chkrootkit detection tests was: `sudo chkrootkit | tee /tmp/chkrootkit.log`.

Checking the system without rootkits installed produced warnings of two suspicious files found in the system: `/usr/lib/debug/build-id` and `/lib/modules/4.15.0-45-generic/vdso/.build-id`. Chkrootkit did not provide any details on why the files were considered suspicious. Also, no summary of the results of the test was given by the tool. A snippet of the clean Chkrootkit detection run is shown in Figure 11.

```
ROOTDIR is `/'
Checking `amd'... not found
Checking `basename'... not infected
Checking `biff'... not found
Checking `chfn'... not infected
Checking `chsh'... not infected
Checking `cron'... not infected
Checking `crontab'... not infected
...
```

Figure 11. First few check results in the Chkrootkit clean run.

Chkrootkit did not explicitly detect any of the injected rootkits. However, it did provide warnings of a potential rootkit installation for both JynxKit2 and Zendar. In the case of JynxKit2, the chkproc check included in Chkrootkit detected a process hidden from the `ps` command with result "Warning: Possible LKM Trojan installed". The same warning was given when testing Zendar, although by the chkdirs check instead of chkproc, and without any additional details of the potential infection.

The execution of Chkrootkit tests were also found to differ from the clean run's results for Azazel, Bedevil and Vlany. When executing a detection run with Azazel injected, Chkrootkit's output included what appears to be the output of the commands it was checking instead of the results of the checks. For example, the check for the `env` command printed the environment variables defined for the shell running Chkrootkit instead of normal results like "not infected" or "not found". The Azazel detection test was re-executed twice to confirm that the abnormal test execution was actually related to Azazel. A snippet of the malformed output logs in the Azazel detection run is shown in Figure 12.

```
ROOTDIR is `/'
Checking `amd'... \c
Checking `basename'... \c
Checking `biff'... \c
Checking `chfn'... \c
Changing the user information for root
Enter the new value, or press ENTER for the default
        Full Name [root]:       Room Number []:         Work Phone []:
        Home Phone []:  Other []:
        Checking `chsh'... \c
Changing the login shell for root
Enter the new value, or press ENTER for the default
        Login Shell [/bin/bash]: Checking `cron'... \c
Checking `crontab'... \c
...
```

Figure 12. Abnormal Chkrootkit output results in the Azazel detection run.

When testing Bedevil and Vlany, the check for "bindshell" produced the following error: "Cannot open netlink socket: Input/output error". The errors were verified to occur only when either rootkit was installed by executing the tests and the clean run two additional times. The detection results for the two rootkits were otherwise identical to the clean run.

Of the user mode rootkits, Chkrootkit produced results identical to the clean run only in BEURK's detection run. In fact, the tool encountered more abnormal detection runs than successful detections in the user mode rootkit tests. Chkrootkit was also unable to detect any traces of infection or suspicious activity for any of the kernel mode rootkits used in this study.

4.6 LKRG

LKRG version 0.7 was downloaded from its official release page hosted by Openwall and then manually compiled. Compiling LKRG produced a kernel object file (`p_lkrg.ko`) which, when inserted to the kernel, would immediately start executing its kernel checking procedures.

The LKRG detection runs were conducted twice, as both pre- and post-injection tests. In pre-injection tests, the LKRG kernel module was inserted into the kernel before rootkit injection. Conversely, rootkits were injected into the system before the LKRG kernel module was inserted in post-injection tests. The reasoning for running both pre- and post-injection tests was to check if there is a difference in LKRG's effectiveness for detecting modifications to the kernel based on if the rootkit is injected before or after LKRG has been initialized.

Figure 13 shows LKRG's initialization and test results for a clean run. The output was identical in detection runs where a rootkit was injected into the system but not detected by LKRG. The message "System is clean!" was repeated approximately every 15 seconds, confirming that LKRG kept executing its checks periodically after it was installed.

```
[1765.948683] p_lkrg: loading out-of-tree module taints kernel.
[1765.951045] [p_lkrg] Loading LKRG...
[1765.954991] Freezing user space processes ... (elapsed 0.001
seconds) done.
[1765.956542] OOM killer disabled.
[1766.386322] [p_lkrg] LKRG initialized successfully!
[1766.386326] OOM killer enabled.
[1766.386326] Restarting tasks ... done.
[1781.509581] [p_lkrg] System is clean!
```

Figure 13. LKRG dmesg output for loading and executing a clean detection run.

In the pre-injection tests, LKRG was able to detect kernel modifications made by all kernel mode rootkits except Keysniffer. Figure 14 shows the most relevant part of result output captured using `dmesg`, where LKRG detected a hidden module (Diamorphine) with its cross-view checking feature.

```
[1793.541744] [p_lkrg] ALERT !!! _RODATA MEMORY BLOCK HASH IS
DIFFERENT - it is [0x637744befeda89a5] and should be
[0xcf790917e9f35a54] !!!
[1793.541748] [p_lkrg] ALERT !!! FOUND LESS[1] MODULES IN MODULE
LIST[67] THAN IN KOBJ[68]
[1793.541754] [p_lkrg] HIDDEN MODULE:
              name[diamorphine] module at addr[000000006e864428]
              module core[0000000044e537f8] with size[0x1000]
              hash[0x6d4e0d584a3b6d15]
[1793.541763] [p_lkrg] !! MOST LIKELY SYSTEM IS HACKED - MODULE WILL
BE DUMPED !! **
[1793.541772] [p_lkrg] ALERT !!! SYSTEM HAS BEEN COMPROMISED -
DETECTED DIFFERENT 2 CHECKSUMS !!!
```

Figure 14. LKRG dmesg detection result for the Diamorphine pre-injection detection run.

In the post-injection tests, LKRG was able to detect modifications to the kernel made by Diamorphine, Nuk3 Ghost, Reptile and Rootfoo Linux Rootkit but not by Honey Pot Bears, Keysniffer, LilyOfTheValley, Puszek or Sutekh. This indicates that the post-injection tests are not as effective in detecting kernel modifications compared to when LKRG is installed prior to injection.

Figure 15 displays a portion of the LKRG detection results for Rootfoo Linux Rootkit, where LKRG's kernel exploit detection feature blocks the use of the UMH (Usermode Helper) feature provided by the kernel to invoke Rootfoo's periodically triggered rootkit executable from the user mode. In addition to detecting an exploit, the exploit detection feature also prevented further execution of the discovered program.

```
[1792.762722] ROOTKIT executing /tmp/rootkit.sh
[1792.762905] [p_lkrg] <Exploit Detection> !!! BLOCKING UMH !!!
[1792.762908] [p_lkrg] <Exploit Detection> Someone is trying to
execute file: [/tmp/rootkit.sh]
[1792.762909] [p_lkrg] <Exploit Detection> --- . ---
```

Figure 15. LKRG dmesg detection result for the Rootfoo Linux Rootkit post-injection detection run.

No user mode rootkits were detected by LKRG. The most plausible reason for this is that the selected user mode rootkits do not attempt to directly modify the kernel's data structures in any way. They rather rely on the `LD_PRELOAD` mechanism for hooking and function purely within the userland. Further tests using rootkits that inject themselves using the `/dev/mem` and `/dev/kmem` device files could potentially provide some interesting results in this regard.

4.7 Summary

Of the 75 detection runs conducted, 28 provided indications of a rootkit or suspicious behaviour, 4 executed abnormally and 43 produced results identical to the clean runs. The results of the rootkit detection tests are summarized in Table 5.

Table 5. Rootkit detection results. **x**: detected explicitly (by name), **!**: detected as a potential rootkit, **?**: detected as suspicious, *: abnormal execution, empty cell: not detected (result identical to clean run).

Rootkit	OSSEC	AIDE	Rootkit Hunter	Chkrootkit	LKRG
Azazel		?	*	*	
Bedevil	?	?		*	
BEURK		?			
JynxKit2	!	?	x	!	
Vlany	?	?		*	
Zendar	!	?	x[1]	!	
Diamorphine			x		!
Honey Pot Bears		?			!
Keysniffer					
LilyOfTheValley					!
Nuk3 Gh0st					!
Puszek	!				!
Reptile	!	?			!
Rootfoo Linux Rootkit			x[2]		!
Sutekh					!

The results of rootkit detection runs for each detection tool are categorized per outcome in Table 6. Explicit detections indicates how many rootkits were specifically named by each tool (Rootkit Hunter's two misidentifications are also counted as explicit detections). Potential rootkits/malware shows how many times each tool was able to provide at least one warning of a potential rootkit or malware infection. Suspicious results indicates how many times each tool successfully produced results which couldn't be labelled as potential infections but could be considered as unusual system behaviour. Abnormal test executions specifies how many times the execution of the test did not proceed normally. False positives shows how many times each tool gave a warning of a rootkit or suspicious behaviour in a clean environment or unrelated to a rootkit. Finally, clean results indicates how many times the detection tool's result was identical with the clean detection run.

[1] Misidentified as SucKIT

[2] Misidentified as Diamorphine

Table 6. Rootkit detection run outcome summary.

	Explicit detections	Potential rootkits/ malware	Suspicious results	Abnormal test executions	False positives	Clean results
OSSEC	0	4	2	0	1	8
AIDE	0	0	8	0	0	7
Rootkit Hunter	4	0	0	1	2	8
Chkrootkit	0	2	0	3	0	10
LKRG	0	8	0	0	0	7
Total	**4**	**14**	**10**	**4**	**3**	**40**

OSSEC detected four of the 15 rootkits as potential rootkits and provided plausible warnings of suspicious system behaviour for two other rootkits. It also consistently provided the same false positive in every detection run, including the clean run. OSSEC required more configuration than the other tools to enable triggering its detection checks manually, as it normally conducts automated periodical system checks with a slow scanning pace to reduce its overall impact on system performance.

AIDE was capable of detecting changes in the file system made by eight of the 15 rootkits. All of the user mode rootkits left traces visible to AIDE, while only two kernel mode rootkits (Honey Pot Bears and Reptile) modified the file system in a way that made them detectable by inspecting AIDE's results. Determining the results for AIDE required manual inspection and comparison of the results and the files manipulated by the rootkits, making the determination of the tool's effectiveness a more difficult and involved process. This also meant that all of AIDE's positive results were marked as suspicious behaviour, as no rootkit-related warnings were (or can be) displayed by the tool.

Rootkit Hunter was able to detect four of the 15 rootkits, all of which were direct detections where a rootkit was named. However, two of the directly detected rootkits, Zendar and Rootfoo Linux Rootkit, were incorrectly identified as SucKIT and Diamorphine, respectively. Rootkit Hunter was not able to provide any indications of suspicious behaviour or potential infection for the rest of the rootkits. The detection run was abnormal for Azazel, where it appears that multiple system checks were skipped. Rootkit Hunter produced two false positives (marked as potential rootkits) in its clean run, which were also shown in all detection runs' results. Of the five tools used, only Rootkit Hunter was able to explicitly detect rootkits by identifying them by name.

Chkrootkit was capable of detecting two of the 15 rootkits (JynxKit2 and Zendar), both of which were user mode rootkits. Its detection run malfunctioned when Azazel was installed in the system, where the test output for its checks was from the commands it checked instead of the tool's own results. Detection runs for Bedevil and Vlany also produced some abnormal results. Chkrootkit did not provide any summary of the detection results, meaning that the logfile needed to be fully inspected to determine if something had been detected by the tool.

LKRG was able to detect potential infection for all kernel mode rootkits except Keysniffer in its pre-injection tests. In the post-injection tests, it was able to detect four

out of the nine kernel mode rootkits (Diamorphine, Nuk3 Ghost, Reptile and Rootfoo Linux Rootkit). This indicates that LKRG is more effective as a proactive tool instead of when a rootkit has already been injected to the system. LKRG did not provide any indications of infection for the user mode rootkits used in this study.

Figure 16 displays the number of positive and negative detections by each tool for the user mode rootkits. A positive detection in this case indicates that a rootkit injection was either explicitly detected, presented as a potential rootkit/malware or the system behaviour was considered suspicious. A negative detection on the other hand indicates that a detection run with a rootkit injected either produced false positives, results identical to the clean run or the run was executed abnormally without any warnings.

AIDE provided the most positive detections for user mode rootkits, in the form of manually verifiable indications of file system manipulation for all six of them. OSSEC was the second most effective detector in this category, with four positive and two negative detections, followed by Rootkit Hunter and Chkrootkit with two positive and four negative detections. LKRG was the least effective in this category, with no detections of user mode rootkits.

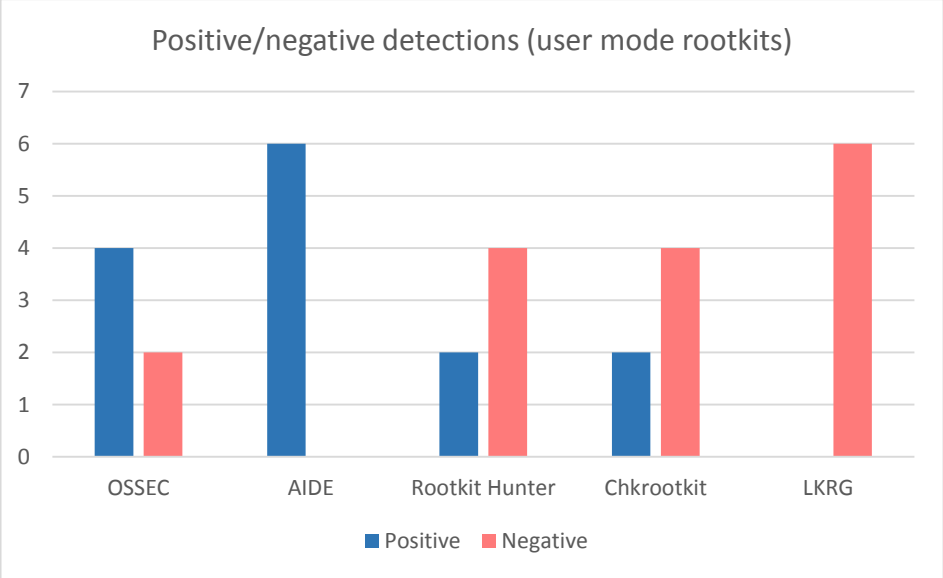

Figure 16. Positive/negative detection indications by tool (user mode rootkits).

Figure 17 shows the number of positive and negative detections by each tool for the kernel mode rootkits. It is clear that LKRG was the most effective for detecting this category of rootkits, with eight positive detections and one negative detection. OSSEC, AIDE and Rootkit Hunter had the same level of effectiveness with two positive and seven negative detections. Chkrootkit did not detect any of the tested kernel mode rootkits, so it was the least effective detection tool in this category.

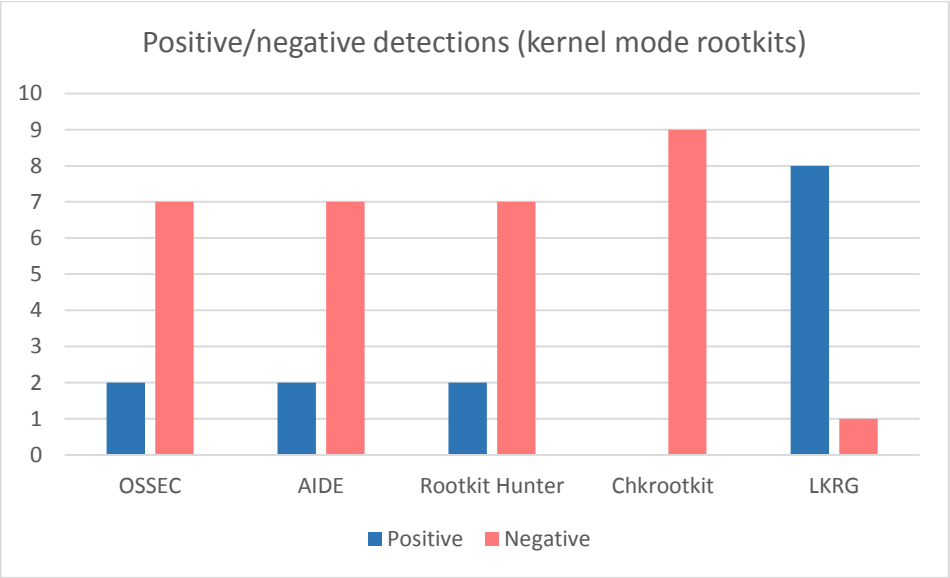

Figure 17. Positive/negative detection indications by tool (kernel mode rootkits).

Figure 18 visualizes the number of positive and negative detections by each tool for the set of all tested rootkits. AIDE and LKRG provided the most positive detections in total, with eight positive and seven negative detections for both tools. The two were also the only detection tools that had more positive than negative detections. OSSEC provided six positive detections and nine detections in total, while Rootkit Hunter provided four positive and 11 negative detections and Chkrootkit provided two positive and 13 negative detections.

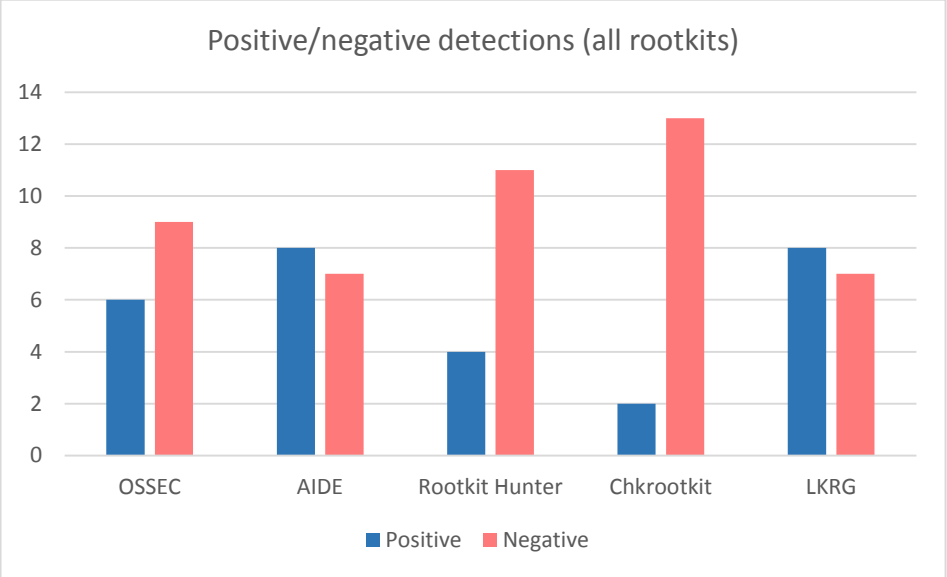

Figure 18. Positive/negative detection indications by tool (all rootkits).

In summary, AIDE was the most effective detection tool for detecting user mode rootkits in terms of positive detection indications, though it required manual inspection and knowledge of the rootkits' behaviour to confirm the infections. LKRG was the most effective detection tool for kernel mode rootkits, especially when it was already running before rootkits were injected. Even when loaded to the kernel after rootkit injection, LKRG was more effective in detecting kernel mode rootkits than the other tools. OSSEC was the second most effective tool in detecting user mode rootkits, and on par with AIDE and Rootkit Hunter in finding kernel mode rootkits. Rootkit Hunter was the

only detection tool capable of explicitly detecting specific rootkits injected to the system. Chkrootkit was the least effective tool in finding user and kernel mode rootkits.

The overall detection effectiveness was less than ideal for individual detection tools, as the tools' average detection rate was 37.3% for all rootkits. The strengths and weaknesses of each tool were noticeable in the results, indicating that the effectiveness of detection could be strategically improved by utilizing multiple tools in detection scenarios. For example, using AIDE and OSSEC for covering user mode rootkits and LKRG for kernel mode rootkits could vastly improve the detection rates, at least for the representation of rootkits used in this study.

In terms of rootkit detectability, Keysniffer was the only rootkit that was not detected by any tool. One potential reason for this is that Keysniffer does not hook the system like the other tested rootkits, as it instead registers a keyboard event notifier with the kernel and does not manipulate e.g. the system call table. It should be noted that some of the stricter rootkit definitions would not consider Keysniffer as a rootkit, though it was selected for this particular study.

Azazel, BEURK, Diamorphine, LilyOfTheValley, Nuk3 Gh0st and Sutekh were only detected once (though Azazel additionally caused two abnormal test executions). The rootkits with most detections were JynxKit2 and Zendar, as they were both detected by four of the five tools. Overall, user mode rootkits exhibited a higher detection rate (46.7%) than kernel mode rootkits (31.1%).

5. Discussion

This chapter summarizes and discusses the findings of this study in relation to prior research. Each research question is considered and addressed individually. Finally, the practicality of the detection tools is discussed.

5.1 Relation to prior studies

The focus of this study was on determining the effectiveness of detection tools in finding indications of Linux rootkits. Similar studies by Bunten (2004) and Todd et al. (2007) have discovered that while such tools show promise in detecting rootkits, they also tend to have shortcomings such as lack of universality and reduced forensic integrity. This study featured tools used in the studies (AIDE, Rootkit Hunter and Chkrootkit) as well as tools for which no similar evaluation of effectiveness was found in prior research (namely OSSEC and LKRG).

Bunten (2004) has called for more general runtime-based detection methods for kernel mode rootkits such as kernel integrity checking, which was featured in this study in the form of LKRG, representing an advancement in the field of such detection tools. Indeed, LKRG displayed promising results in discovering indications of kernel mode rootkits. In addition, the inclusion of OSSEC provided detection results from a tool with a more universal range of detection features, matching four of the five rootkit detection categories described by Todd et al. (2007): signature-, behaviour-, cross-view- and integrity-based detection.

Freiling and Schwittay (2007) have advocated performing periodic evaluations of rootkit detection tools to determine their detection abilities and overall effectiveness. This study provided such an evaluation in the context of modern publicly available Linux rootkits and contemporary open source detection tools. The findings contribute to the body of knowledge by improving the understanding of how effective Linux rootkit detection tools are in general as well as in relation to each other.

5.2 RQ1: Rootkit detection effectiveness

The results of the 75 detection tests conducted in this study showed that the rootkit detection tools provided direct indications of rootkits in four cases, or 5.3% of all detection attempts. Indications of potential rootkit or malware infections were provided by the tools in 14 cases (18.7%). Indications of suspicious system behaviour deviating from normal operation were provided by the tools in 10 detection tests (13.3%).

In total, any indication of infection or suspicious system behaviour was provided in 28 of the detection tests, meaning that even the most optimistic combined detection rate was 37.3%. This means that 62.7% of the detection tests produced results that were consistent with the clean tests, where no rootkits were injected to the system. In other words, no indications of rootkits were found in close to two thirds of the tests.

These results indicate a lower detection effectiveness compared to detection results by Freiling and Schwittay (2007) in their evaluation Windows rootkits and detection tools, where either all or some of the modifications made by the rootkits were discovered in 49 of 99 detection tests, for a combined detection rate of 49.5%. It should be noted that Freiling and Schwittay (2007) did not run detection tests for every combination of rootkit and detection tool, as some of the tools were not applicable for detecting indications of some of the rootkits' hiding functionalities. Therefore, to facilitate a more straightforward comparison, the inapplicable tests were counted (by the author of this study) as ones where no rootkit indications were detected.

There was also a significant difference in the detection rate for user mode and kernel mode rootkits. User mode rootkits had a combined detection rate of 46.7%, while for kernel mode rootkits the rate was 31.1%. This shows indications that the detection tools are on average more likely to detect user mode rootkits than kernel mode rootkits. However, it should be kept in mind that the detection rates can be drastically different for a different selection of rootkits. The detection rates for each detection tool are summarized in Table 7.

Table 7. Detection rates per detection tool.

	All rootkits	**User mode rootkits**	**Kernel mode rootkits**
OSSEC	40%	66.7%	22.2%
AIDE	53.3%	100%	22.2%
Rootkit Hunter	26.7%	33.3%	33.3%
Chkrootkit	13.3%	33.3%	0%
LKRG	53.3%	0%	88.9%
Average	**37.3%**	**46.7%**	**31.1%**

The results suggest that the rootkit detection tools' effectiveness in detecting modern Linux rootkits is rather limited when used individually. However, using a combination of tools with different detection approaches could significantly improve the overall detection effectiveness. For example, using a combination of OSSEC, AIDE and LKRG would find indications of 93.3% of the rootkits (all except Keysniffer) used in this study. This result is consistent with the findings of Freiling and Schwittay (2007), who advocated the use of a combination of multiple rootkit detection tools for more effective and reliable detection results.

5.3 RQ2: Differences in detection tool effectiveness

The rootkit detection tools used in this study had some notable differences in terms of effectiveness. Rootkit Hunter was the only detection tool that was able to directly detect any rootkits, with four rootkits explicitly named (two of which were misidentifications). AIDE and LKRG provided the most combined indications of potential rootkit infections and suspicious system behaviour, with eight detection tests for both tools providing non-negative results.

AIDE was the most effective in detecting indications of user mode rootkits, as its results showed modifications made to the file system by all of them. However, it should be

noted that the detection results for AIDE required further manual analysis and could thus produce variability in future studies with different analysis methods and rootkit sample sets. As AIDE only shows the additions, deletions and modifications to files and directories in the file system, it does not provide direct warnings specific to rootkits. This means that the effectiveness of detection using AIDE relies on the investigator's knowledge of the rootkits' behaviour and characteristics and may not be practical in most scenarios where such intimate knowledge is not available.

LKRG was by far the most effective tool in detecting kernel mode rootkits, providing indications of infection for 8 of the 9 rootkits, for a kernel mode rootkit detection rate of 88.9%. This is in stark contrast with the other detection tools, which had an average kernel mode rootkit detection rate of 21.4%. The result appears to validate Bunten's (2004) advocation for future detection tools relying on kernel integrity checking to increase their overall effectiveness. However, LKRG was not able to detect any user mode rootkits. This can be explained by LKRG's inherent focus on checking the integrity of the kernel's data structures for modifications, which was not part of the feature set of any user mode rootkit used in this study.

Chkrootkit did not detect any kernel mode rootkits and had the lowest combined detection rate of all detection tools, at 13.3%. This result supports the finding by Todd et al. (2007), where Rootkit Hunter was found to be more effective out of the two tools in detecting rootkits, especially when considering the explicit detections provided by Rootkit Hunter. However, Rootkit Hunter still had the second lowest overall detection rate at 26.7%, meaning that its effectiveness in finding the rootkits used in this particular study wasn't exceptional.

OSSEC was the second most effective tool in detecting user mode rootkits with a 66.7% detection rate, though it did not fare as well in detecting kernel mode rootkits, with a detection rate of 22.2%. It provided one false positive result, while Rootkit Hunter provided two false positives denoting rootkit infection in cases where no rootkits were injected to the system. The false positives were consistent throughout all detection tests, meaning that they can potentially be ruled out and predicted by researchers and other users conducting similar investigations.

5.4 RQ3: Detection of rootkit types

The results of the detection tests show that both user mode and kernel mode rootkits can be detected by the set of detection tools used in this study. As rootkits that operate below the kernel level (in protection rings -1, -2 and -3) were not included in this study, the applicability of the detection tools for such rootkits couldn't be verified. However, it is unlikely that the included detection tools would be effective against such rootkits, as their actions may not be at all visible to the user space or the kernel, which the detection tools solely rely on for information.

The results also suggest that rootkits incorporating hiding, privilege escalation and/or backdoor functionalities can be detected by the detection tools. The hiding features appeared to be especially susceptible to detection, as OSSEC, AIDE and Chkrootkit managed to provide positive indications of rootkits primarily based on hidden files. However, not all included rootkits were detected by the tools, as no detection tool was able to detect Keysniffer or even provide any form of indication that differed from the results of the clean detection runs.

A likely reason for why Keysniffer managed to avoid detection is that it does not manipulate the system call table or other kernel data structures in a way that makes its behaviour particularly suspicious. Instead, it registers a notifier for keyboard events in a way that can be difficult to distinguish from other, more legitimate kernel modules. The result closely mirrors that of a similar study by Freiling and Schwittay (2007), where a keyboard sniffer rootkit (Klog) was not detected at all because it did not include any hiding functionality. It is important to note that Keysniffer indeed does not meet all of the criteria of more strict definitions of rootkits, and it may be problematic to use in comparisons where such definitions are used.

5.5 Practicality of the detection tools

As the effectiveness of detection is often limited by the investigators' resources (Bunten, 2004), the practicality of rootkit detection tools is also a relevant property to consider. In terms of practicality and general usability, there were differences between the detection tools, as some required more configuration and manual intervention than others. However, as the level of practicality can be heavily dependent on the knowledge and skillset of the person conducting rootkit detection, quantifying it unambiguously is not a trivial task.

AIDE's results required the most interpretation, as the tool does not attempt to provide warnings or other notifications to the user beyond notifying whether or not the contents of the file system have changed since the most recent integrity check. This means that detecting rootkits using AIDE is always dependent on the user's knowledge and skills in "connecting the dots" between rootkit behaviour and the state of the file system's contents. Thus, its practicality is limited for example in scenarios where the detection process is automated or where the user does not possess sufficient knowledge of rootkit behaviour.

LKRG was relatively simple to setup, as while it had to be manually downloaded and compiled, it did not require any additional configuration. Once the LKRG kernel module was running, it periodically (or upon detection) sent the results of its checks to the kernel message buffer, readable with the `dmesg` command. While arduous to read alongside other kernel events, the results themselves were concise and readable. However, the lack of any notification features and simple user-space controls makes LKRG less than optimal for less advanced users.

OSSEC required the most configuration to setup for the detection runs. Setting up the latest version of the tool required manual compilation, while ensuring that the checks were run on demand (and within specified time constraints) required configuration of some internal variables in configuration files. However, this was not the more common usage scenario for the tool, where checks are conducted periodically and in small batches to reduce the tool's impact in system performance. OSSEC features extensive alert notification options and its result output was detailed yet comprehensible, making it suitable for continuous monitoring purposes.

Rootkit Hunter and Chkrootkit were among the simplest tools to use, as they could be directly installed using Ubuntu's APT package manager and did not require extra configuration (except for the initialization of Rootkit Hunter's file property database). Both tools were invoked directly from the command line and sent their output to both standard output and a logfile. Rootkit Hunter produced a useful summary at the end of its detection run, indicating if any signs of rootkits, malware or suspicious behaviour

were detected. Chkrootkit did not, requiring its full, though brief, output to be manually inspected to uncover signs of infection. Rootkit Hunter contains functionality to send warnings over email, making it more suitable and convenient for continuous monitoring.

It should be noted that none of the detection tools provide any recommendations or guidance in the case of a positive detection result. Thus, the task of removing rootkits from the system is completely left for the user to handle. LKRG was the only tool that could prevent rootkit code from executing in certain situations, making it potentially valuable for any system where immediate reactive action to kernel rootkit injection and operations is desired.

Compared to manual detection procedures, the detection tools are less flexible, though they do not require as much in-depth knowledge of rootkits and operating system internals. While conducting a memory forensics analysis of rootkits, such as the one described by Carbone (2014), would arguably be much more thorough and likely to discover well-hidden rootkits, doing so would be much less practical in scenarios where the rootkit infection and its specifics are not known. Therefore, using automated detection tools is a vastly more practical solution for most rootkit detection use cases.

6. Conclusion

The purpose of this study was to evaluate the effectiveness of rootkit detection tools in the context of the Linux operating system. Five detection tools – OSSEC, AIDE, Rootkit Hunter, Chkrootkit and LKRG – were evaluated by conducting a series of 75 detection tests with 15 different rootkits. Each individual test consisted of injecting a publicly available rootkit to a virtual machine environment running Ubuntu 16.04, executing a detection tool and gathering data provided by the tool once it had finished running. The tests were conducted for every combination of rootkit detection tool and rootkit used in this study. Once all of the tests were finished, the collected data was analysed to determine what indications of rootkit infection (if any) each detection tool was able to report in their results.

The results showed that only 37.3% of the detection tests were able to either explicitly detect a rootkit, provide indications of potential rootkit infection or warn about suspicious system behaviour. The rest of the detection tests failed to produce any evidence of rootkit infection or other activity differing from normal system behaviour. However, using a combination of detection tools increased the overall detection rate significantly, as every rootkit except for one (Keysniffer) was discovered by at least one detection tool.

Effectiveness of the rootkit detection tools used in this study was mixed, with overall detection rates varying from 13.3% to 53.3%. There was also variation between detection rates for the two categories of rootkits (user mode and kernel mode) used in this study, as the overall detection rate was 46.7% for user mode rootkits and 31.1% for kernel mode rootkits. This also showed that both rootkit categories could be detected by at least some of the detection tools. Overall, the results showed that while effectiveness for an individual detection tool could be suboptimal, using multiple tools vastly increased the probability of detection.

The findings of this study can be useful for both practitioners attempting to maximize the effectiveness of their rootkit and malware detection systems and improving the overall security of their computer systems, as well as academics conducting research on rootkits and rootkit detection. This study also contributes to the body of knowledge with documentation of rootkit injection and verification procedures, description of a systematic testing process and reporting of detection results from a variety of rootkits and rootkit detection tools.

One limitation of this study was that the detection tests were conducted in a virtual machine environment. The emulated environment can be detected by rootkits (and other software), which allows them to adjust their behaviour to complicate potential forensic analysis procedures and to compromise detection results. However, no indication of such adjustments occurring without manual action could be found in a cursory audit of the rootkits' source code.

The rootkits selected for this study were all open source software and publicly available, leading to potential selection bias where the more general set of rootkits is not accurately represented. No malicious rootkits were collected using research honeypots

or other collection mechanisms, as doing so was considered too time consuming and potentially unsafe in the scope of this study. Including only publicly available rootkits could impact the generalizability of the results, as the rootkits' functionalities and operating mechanisms can be freely studied in detail by authors of the detection tools and other parties to improve the detectors' abilities to detect them as well as other rootkits using similar mechanisms and techniques. Also, it is unclear how comparable the techniques used by publicly available rootkits are to "real-world" malicious rootkits used by malicious actors. Further research could take this into account by using malicious rootkits captured in the wild in addition to those that are publicly available.

Future studies focusing on the evaluation of rootkit detection effectiveness could expand the scope of detection tools to cover detection contexts outside of host-based detection, where detectors would not exist in the same environment as the rootkits themselves. For example, a study where tools using network-based detection and virtual machine introspection (VMI) are evaluated would help in understanding how exactly the environment in which the detector is placed can affect the effectiveness of rootkit detection.

Another potential avenue of future research would be to conduct a more in-depth comparison of the effectiveness of the specific detector categories the detection tools can be classified as. Such studies could provide valuable insight into how effective each category can actually be, what their pros and cons are and what kinds of rootkits they can be most applicable to. This, in turn, could help in improving the effectiveness of rootkit detectors themselves.

Similar studies evaluating rootkit detection effectiveness could also be conducted in the future to determine how more modern rootkits are able to evade detection and how effective contemporary and future detectors are in detecting such rootkits. Such evaluations would help in retaining up-to-date knowledge on rootkit detection effectiveness. The results of such studies could also aid in understanding and giving recommendations on which detection tools to rely on for effective rootkit detection.

References

Algayar, M. (2017). LilyOfTheValley [Software]. Retrieved 31.1.2020 from https://github.com/En14c/LilyOfTheValley

Augusto, I. (2019). Reptile [Software]. Retrieved 8.12.2019 from https://github.com/f0rb1dd3n/Reptile

Bierfert, A. (2007). Seminar: Advanced Exploitation Techniques. Aachen, DE: RWTH Aachen University.

Bravo, P., & Garcia, D. F. (2011). Proactive Detection of Kernel-mode Rootkits. In *2011 Sixth International Conference on Availability, Reliability and Security*, pp. 515-520. IEEE.

Bunten, A. (2004). Unix and Linux Based Rootkits Techniques and Countermeasures. In *16th Annual First Conference on Computer Security Incident Handling*, Budapest.

Carbone, R. (2014). Malware Memory Analysis of the Jynx2 Linux Rootkit (Part 1): Investigating a Publicly Available Linux Rootkit Using the Volatility Memory Analysis Framework. Defence Research and Development Canada.

Chokepoint (2017). Azazel [Software]. Retrieved 8.12.2019 from https://github.com/chokepoint/azazel

Cisco (2019). ClamAV [Software]. Retrieved 7.2.2020 from https://www.clamav.net/

En14c (2017). LilyOfTheValley [Software]. Retrieved 28.1.2020 from https://github.com/En14c/LilyOfTheValley

ErrProne/BlackHatAcademy.org (2011). JynxKit [Software]. Retrieved 8.1.2020 from https://github.com/chokepoint/jynxkit

ErrProne/BlackHatAcademy.org (2011). JynxKit2 [Software]. Retrieved 8.12.2019 from https://github.com/chokepoint/Jynx2

Eterna1 (2018). Puszek [Software]. Retrieved 8.12.2019 from https://github.com/Eterna1/puszek-rootkit

Free Software Foundation (2015). ld.so, ld-linux.so - dynamic linker/loader. Retrieved 6.1.2020 from https://linux.die.net/man/8/ld.so

Freiling, F. C., & Schwittay, B. (2007). Towards Reliable Rootkit Detection in Live Response. In *3rd International Conference on IT Incident Management & IT Forensics*, pp. 125-144.

Garfinkel, T., & Rosenblum, M. (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS 3*, pp. 191-206.

Gustedt, J., Jeannot, E., & Quinson, M. (2009). Experimental Methodologies for Large-scale Systems: A Survey. *Parallel Processing Letters*, 19(03), pp. 399-418.

Harley, D., & Lee, A. (2007). The Root of All Evil? - Rootkits Revealed [White paper]. Retrieved 6.1.2010 from http://eset.version-2.sg/softdown/manual/Whitepaper-Rootkit_Root_Of_All_Evil.pdf

Henderson, B. (2006). Linux Loadable Kernel Module HOWTO. Retrieved 6.1.2010 from https://www.tldp.org/HOWTO/pdf/Module-HOWTO.pdf

Hodges, M. (2019). Linux Rootkit [Software]. Retrieved 31.1.2020 from https://github.com/rootfoo/rootkit

Hoglund, G., & Butler, J. (2006). *Rootkits: Subverting the Windows Kernel*. Boston, MA: Addison-Wesley Professional.

Hua, Q., & Zhang, Y. (2015). Detecting Malware and Rootkit Via Memory Forensics. In *2015 International Conference on Computer Science and Mechanical Automation (CSMA)*, pp. 92-96. IEEE.

Hudson, T., & Rudolph, L. (2015). Thunderstrike: EFI Firmware Bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, p. 15. ACM.

Hudson, T., Kovah, X., & Kallenberg, C. (2015). Thunderstrike 2: Sith Strike. Black Hat USA Briefings. Retrieved 21.2.2020 from https://www.blackhat.com/docs/us-15/materials/us-15-Hudson-Thunderstrike-2-Sith-Strike.pdf

Hyde, D. (2009). A Survey on the Security of Virtual Machines. St. Louis, MO: Washington University in St. Louis. Retrieved 9.2.2020 from https://www.cse.wustl.edu/~jain/cse571-09/ftp/vmsec.pdf

Jana, A. P. (2019). Keysniffer [Software]. Retrieved 8.12.2019 from https://github.com/jarun/keysniffer

Kim, S., Park, J., Lee, K., You, I., & Yim, K. (2012). A Brief Survey on Rootkit Techniques in Malicious Codes. *Journal of Internet Services and Information Security (JISIS)*, 2(3/4), pp. 134-147.

Kleiman, I., Gao, J., Khan, I. & Song, D. (2019). Honey Pot Bears Rootkit [Software]. Retrieved 28.1.2020 from https://github.com/shortland/Honey-Pot-Bears-Rootkit

Lehti, R., & Virolainen, P. (2019). AIDE - Advanced Intrusion Detection Environment [Software]. Retrieved 31.1.2020 from https://github.com/aide/aide

Love, R. (2010). *Linux Kernel Development* (3rd ed.). Boston, MA: Addison-Wesley Professional.

R-fx Networks (2019). Linux Malware Detect [Software]. Retrieved 7.2.2020 from https://www.rfxn.com/projects/linux-malware-detect/

Mello, V.R. (2019). Diamorphine [Software]. Retrieved 8.12.2019 from https://github.com/m0nad/Diamorphine

Mempodippy (2019). Vlany [Software]. Retrieved 8.12.2019 from https://github.com/mempodippy/vlany

Murilo, N. & Steding-Jessen, K. (2019). Chkrootkit [Software]. Retrieved 24.11.2019 from http://www.chkrootkit.org/

Schales, D.L., Hess, D., Warraich, K. & Safford, D (2019). Tiger: The Unix Security Audit and Intrusion Detection Tool [Software]. Retrieved 7.2.2020 from http://nongnu.org/tiger/

Tripwire (2019). Open Source Tripwire [Software]. Retrieved 7.2.2020 from https://github.com/Tripwire/tripwire-open-source

O'Neill, R. E. (2016). *Learning Linux Binary Analysis*. Birmingham, UK: Packt Publishing.

Openwall (2019). LKRG – Linux Kernel Runtime Guard [Software]. Retrieved 7.2.2020 from https://openwall.info/wiki/p_lkrg/Main

Oracle (2020). VirtualBox User Manual. Retrieved 14.2.2020 from https://www.virtualbox.org/manual/ch01.html

OSSEC Project Team (2019). OSSEC [Software]. Retrieved 24.11.2019 from https://www.ossec.net/

Petroni Jr, N. L., Fraser, T., Molina, J., & Arbaugh, W. A. (2004). Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, pp. 179-194.

PinkP4nther (2019). Sutekh [Software]. Retrieved 8.12.2019 from https://github.com/PinkP4nther/Sutekh

Richer, T. J., Neale, G., & Osborne, G. (2015). On the Effectiveness of Virtualisation Assisted View Comparison for Rootkit Detection. In *Proceedings of the 13th Australasian Information Security Conference (AISC 2015)*, 27, p. 30.

Ring-1 (2015). Zendar [Software]. Retrieved 28.1.2020 from https://github.com/ring-1/zendar

Rootkit Hunter (2018). The Rootkit Hunter Project [Software]. Retrieved 24.11.2019 from http://rkhunter.sourceforge.net/

Rowan, J. (2019). Bedevil [Software]. Retrieved 8.12.2019 from https://github.com/naworkcaj/bdvl

Russinovich, M. (2006a). Rootkits in Commercial Software. Retrieved 24.11.2019 from: https://blogs.technet.microsoft.com/markrussinovich/2006/01/15/rootkits-in-commercial-software

Russinovich, M. (2006b). RootkitRevealer [Software]. Retrieved 21.1.2020 from: https://docs.microsoft.com/en-us/sysinternals/downloads/rootkit-revealer

Schällibaum, J. (2018). Nuk3 Gh0st [Software]. Retrieved 31.1.2020 from: https://github.com/ropch4ins/Nuk3Gh0stBeta

Singh, B., Evtyushkin, D., Elwell, J., Riley, R., & Cervesato, I. (2017). On the Detection of Kernel-level Rootkits Using Hardware Performance Counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 483-493. ACM.

Tian, Z., Wang, B., Zhou, Z., & Zhang, H. (2011). The Research on Rootkit for Information System Classified Protection. In *2011 International Conference on Computer Science and Service System (CSSS)*, pp. 890-893. IEEE.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*, pp. 191-194. London, UK: Chapman and Hall/CRC.

Tereshkin, A. & Wojtczuk, R. (2009). Introducing Ring -3 Rootkits [PowerPoint slides]. Retrieved 6.1.2020 from https://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf

Todd, A., Benson, J., Peterson, G., Franz, T., Stevens, M., & Raines, R. (2007). Analysis of Tools for Detecting Rootkits and Hidden Processes. In *IFIP International Conference on Digital Forensics*, pp. 89-105. Springer, New York, NY.

Unix-Thrust (2017). BEURK [Software]. Retrieved 8.12.2019 from https://github.com/unix-thrust/beurk

Wampler, D., & Graham, J. (2007). A Method for Detecting Linux Kernel Module Rootkits. In *IFIP International Conference on Digital Forensics*, pp. 107-116. Springer, New York, NY.

Wichmann, R. (2006). The SAMHAIN file integrity/host-based intrusion detection system [Software]. Retrieved 24.11.2019 from https://www.la-samhna.de/samhain/index.html

Appendix A. Literature search terms and results

Google Scholar (patents and citations excluded):

Search term(s)	**Total results**	**Relevant results**
“rootkit detection” OR “rootkit detector” OR “rootkit scanner”	1520	85
("rootkit detection" OR "rootkit detector" OR “rootkit scanner”) AND (linux OR unix OR bsd) AND "effectiveness"AND "evaluation"	391	30
"linux rootkit" OR "rootkit for linux"	167	40
("linux rootkit" OR "rootkit for linux") AND (detection OR detector OR scanner)	150	30
“unix rootkit” OR “rootkit for unix”	14	10
“bsd rootkit” OR “rootkit for bsd”	1	1
“windows rootkit” OR “rootkit for windows”	486	30
("rootkit detection" OR "rootkit detector" OR “rootkit scanner”) AND (rkhunter OR “rootkit hunter” OR chkrootkit OR samhain OR ossec)	302	31
("rootkit detection" OR "rootkit detector" OR “rootkit scanner”) AND (rkhunter OR “rootkit hunter” OR chkrootkit OR samhain OR ossec) AND “effectiveness” AND “evaluation” AND “linux”	83	14
“rootkit” AND “linux” AND (jynxkit OR jynxkit2 OR azazel OR beurk OR vlany OR bedevil OR diamorphine OR keysniffer OR nurupo OR puszek OR reptile OR rkduck OR “rootkit-kernel-module” OR rtkit OR sutekh OR wukong)	70	16

Microsoft Academic:

Search term(s)	**Total results**	**Relevant results**
rootkit detection	199	35
rootkit detector	604	61
linux rootkit detection	149	52
línux rootkit detector	40	14
linux rootkit	36	20
unix rootkit	994	33
bsd rootkit	972	45
windows rootkit	77	27
chkrootkit	0	0
rkhunter, rootkit hunter	0	0
ossec	12	3
samhain	19	2

CiteSeerX:

Search term(s)	**Total results**	**Relevant results**
“rootkit detection”	2321	22
“rootkit detector”	654	8
“linux rootkit”	1797	17
“unix rootkit”	1164	12
“bsd rootkit”	517	7
“windows rootkit”	1922	17
chkrootkit	108	5
rkhunter	19	2
ossec rootkit	58	5
samhain rootkit	46	3

IEEE Xplore, ACM Digital Library, Wiley Online Library, arXiv & SpringerLink (results represented for each database as *relevant*/*total*):

Search term(s)	**IEEE**	**ACM**	**Wiley**	**arXiv**	**SpringerLink**
“rootkit detection” OR “rootkit detector” OR “rootkit scanner”	18/39	13/76	1/4	7/13	8/217
"linux rootkit" OR "rootkit for linux"	9/49	3/24	1/3	0/2	3/18
"linux rootkit” AND detection	2/28	2/14	0/3	0/2	3/14
rkhunter OR “rootkit hunter” OR chkrootkit OR samhain OR ossec	1/14	0/0	1/5	0/1	7/95
rootkit AND (jynxkit OR jynxkit2 OR azazel OR beurk OR vlany OR bedevil OR diamorphine OR keysniffer OR nurupo OR puszek OR reptile OR rkduck OR “rootkit-kernel-module” OR rtkit OR sutekh OR wukong)	0/0	0/0	0/0	0/0	1/7

Appendix B. Detection tool setup

Before detection tool setup and the clean (stage 1) snapshot, the following commands were executed:

```
# 1. disable unattended-upgrades and ntp
sudo systemctl disable --now unattended-upgrades
sudo timedatectl set-ntp 0

# 2. install common tools
sudo apt install -y build-essential openssh-client openssh-server
```

OSSEC:

```
# 1. install compilation dependencies
sudo apt install -y libevent-dev libpcre2-dev libsqlite3-dev libssl-
dev sendmail zlib1g-dev

# 2. download, unpack and install OSSEC
OSSEC_VERSION=3.5.0
wget https://github.com/ossec/ossec-hids/archive/$OSSEC_VERSION.tar.gz
tar xf $OSSEC_VERSION.tar.gz
cd ossec-hids-$OSSEC_VERSION

export USER_LANGUAGE=en \
    USER_INSTALL_TYPE=local \
    USER_DIR=/var/ossec \
    USER_ENABLE_SYSCHECK=y \
    USER_ENABLE_ROOTCHECK=y \
    USER_ENABLE_EMAIL=n \
    USER_ENABLE_ACTIVE_RESPONSE=n \
    USER_UPDATE=n \
    USER_UPDATE_RULES=n \
    USER_CLEANINSTALL=y

echo | sudo -E ./install.sh

# 3. configure OSSEC for testing
# syscheck and rootcheck: disable sleeping between scans
sed -re 's/(sys|root)(check.sleep)=2/\1\2=0/' -i \
    /var/ossec/etc/internal_options

# rootcheck: scan all files
sed -e 's/<rootcheck>/&\n<scanall>yes<\/scanall>/' -i \
    /var/ossec/etc/ossec.conf

# 4. capture VM snapshot - Ubuntu 16.04.6 :: ossec installed


# 5. execute OSSEC detection (after rootkit injection)
OSSEC_DIR="/var/ossec"
AGENT_ID="000"

$OSSEC_DIR/bin/syscheck_control -u $AGENT_ID
$OSSEC_DIR/bin/rootcheck_control -u $AGENT_ID
```

```
$OSSEC_DIR/bin/ossec-control start
$OSSEC_DIR/bin/agent_control -r -u $AGENT_ID

# 6. capture OSSEC results logs (after detection run)
$OSSEC_DIR/bin/syscheck_control -i $AGENT_ID > /tmp/syscheck.log
$OSSEC_DIR/bin/rootcheck_control -i $AGENT_ID > /tmp/rootcheck.log
```

AIDE:

```
# 1. install AIDE
sudo apt install aide

# 2. initialize AIDE database and generate configuration file
sudo aideinit
mv /var/lib/aide/aide.db.new /var/lib/aide/aide.db
sudo update-aide.conf
mv /var/lib/aide/aide.conf.autogenerated /etc/aide/aide.conf

# 3. update AIDE database
sudo aide -c /etc/aide/aide.conf --update

# 4. capture VM snapshot - Ubuntu 16.04.6 :: aide installed

# 5. execute AIDE detection (after injection), capture results
sudo aide -c /etc/aide/aide.conf --check 2>&1 | tee /tmp/aide.log
```

Rootkit Hunter:

```
# 1. install rkhunter
sudo apt install rkhunter

# 2. update file property database
sudo rkhunter --propupd

# 3. capture VM snapshot - Ubuntu 16.04.6 :: rkhunter installed

# 4. execute rkhunter detection (after injection)
sudo rkhunter --check --sk
```

Chkrootkit:

```
# 1. install chkrootkit
sudo apt install chkrootkit

# 2. capture VM snapshot - Ubuntu 16.04.6 :: chkrootkit installed

# 3. execute chkrootkit detection (after injection), capture results
sudo chkrootkit 2>&1 | tee /tmp/chkrootkit.log
```

LKRG:

```
# 1. download and compile LKRG
LKRG_VERSION=0.7
wget https://openwall.com/lkrg/lkrg-$LKRG_VERSION.tar.gz
tar xf lkrg-$LKRG_VERSION.tar.gz
cd lkrg-$LKRG_VERSION
make

# 2. capture VM snapshot - Ubuntu 16.04.6 :: lkrg installed (post)

# 3. load LKRG to kernel (before injection in pre-injection tests,
     after injection in post-injection tests)
sudo insmod p_lkrg.ko

# 4. capture VM snapshot - Ubuntu 16.04.6 :: lkrg installed (pre)
```

Appendix C. Rootkit setup

Before rootkit setup, the following command was executed to install common build dependencies:

```
sudo apt install bison flex git libpcap-dev libpam0g-dev libssl-dev
```

Azazel:

```
# 1. download and install azazel
git clone https://github.com/chokepoint/azazel
cd azazel
make
sudo make install

# 2. capture VM snapshot - Ubuntu 16.04.6 :: azazel injected

# 3. verify injection
touch test__file.txt; echo $?
> 1
```

Bedevil:

```
# 1. download and install bedevil
git clone https://github.com/naworkcaj/bdvl
cd bdvl
chmod +x bedevil.sh
sudo ./bedevil.sh -i
    BD_UNAME = TyBOtUqK
    BD_PWD = BhhdSqAj
    BD_ENV = kXQMuFnB
    IDIR = /usr/share/doc/libcolord2/.19465
    PAM_PORT = 3777
    MGID = 1256

# 2. capture VM snapshot - Ubuntu 16.04.6 :: bedevil injected

# 3. verify injection
echo hello world > test_file.txt
cat test_file
> hello world
sudo chown 0:1256 test_file
cat test_file; echo $?
> 1
```

BEURK:

```
# 1. download and install beurk
git clone https://github.com/unix-thrust/beurk
cd beurk
make
sudo make infect

# 2. capture VM snapshot - Ubuntu 16.04.6 :: beurk injected

# 3. verify injection
touch test_BEURK_file.txt; echo $?
> 1
```

JynxKit2:

```
# 1. download and install jynxkit2
git clone https://github.com/chokepoint/jynx2
cd jynx2
head -n-2 config.h > config.h
./packer.sh
chmod +x autokitter.sh
sudo ./autokitter.sh

# 2. capture VM snapshot - Ubuntu 16.04.6 :: jynxkit2 injected

# 3. verify injection
# bash tab completion for /etc/ld.so.preload shows file exists
ls /etc/ld.so.preload; echo $?
> 2
```

Vlany:

```
# 1. download and install vlany
git clone https://github.com/mempodippy/vlany
cd vlany
sudo ./install.sh
    Bootloader location = /boot/grub/grub.cfg
    PAM backdoor user = hello
    PAM backdoor password = hello
    PAM backdoor port = 6699
    accept() backdoor encryption = no
    accept() shell password = hello
    accept() low port = 826
    accept() high port = 829
    execve() command password = hello
    Rootkit library name = IZ0CpjjalYvh
    Hidden directory = /lib/libc.so.hello.03
    Environment variable = RNSCWOYUBQIG
    Autoremove directory = no
    Snodew backdoor = no


# 2. capture VM snapshot - Ubuntu 16.04.6 :: vlany injected

# 3. verify injection
# bash tab completion for /lib/libc.so.hello.03 shows file exists
ls /lib/libc.so.hello.03; echo $?
> 2
```

Zendar:

```
# 1. download and install zendar
git clone https://github.com/ring-1/zendar
cd zendar

# in line 74 of file 'install', move -ldl to just after zendar.o to
fix linker flag ordering and enable successful compilation
sudo ./install

# 2. capture VM snapshot - Ubuntu 16.04.6 :: zendar injected

# 3. verify injection
touch test_zendar.txt; echo $?
> 1
```

Diamorphine:

```
# 1. download and install diamorphine
git clone https://github.com/m0nad/diamorphine
cd diamorphine
make
sudo insmod diamorphine.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: diamorphine injected

# 3. verify injection
time cat &
> [1] 9590
> [1]+ Stopped    time cat
ps o cmd 9590; echo $?
> 0
kill -s 31 9590
ps o cmd 9590; echo $?
> 1
kill -s 31 9590
ps o cmd 9590; echo $?
> 0
fg
> time cat
```

Honey Pot Bears Rootkit:

```
# 1. download and install honey-pot-bears-rootkit
git clone https://github.com/shortland/honey-pot-bears-rootkit
cd honey-pot-bears-rootkit
make
sudo insmod notarootkit.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: honey pot bears injected

# 3. verify injection
touch secret
ls | grep secret; echo $?
> 1
```

Keysniffer:

```
# 1. download and install keysniffer
git clone https://github.com/jarun/keysniffer
cd keysniffer
make
sudo insmod kisni.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: keysniffer injected

# 3. verify injection
ls /sys/kernel/debug/kisni/keys; echo $?
> 0
```

LilyOfTheValley:

```
# 1. download and install lilyofthevalley
git clone https://github.com/en14c/lilyofthevalley
cd lilyofthevalley
make
sudo insmod lilyofthevalley.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: lilyofthevalley injected

# 3. verify injection
touch lilyofthevalley_test.txt
ls | grep lilyofthevalley_test.txt; echo $?
> 1
```

Nuk3 Gh0st:

```
# 1. download and install nuk3gh0stbeta
git clone https://github.com/ropch4ins/nuk3gh0stbeta
cd nuk3gh0stbeta
make
sudo insmod rootkit.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: nuk3 gh0st injected

# 3. verify injection
ls | grep README.md; echo $?
> 0
./client --hide-file README.md
ls | grep README.md; echo $?
> 1
```

Puszek:

```
# 1. download and install puszek
git clone https://github.com/eterna1/puszek-rootkit
cd puszek-rootkit
make
sudo insmod rootkit.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: puszek injected

# 3. verify injection
touch test.rootkit
ls | grep test.rootkit; echo $?
> 1
```

Reptile:

```
# 1. download and install reptile
git clone https://github.com/f0rb1dd3n/reptile
cd reptile
sudo ./setup.sh install
    Hide name = reptile
    Auth token to magic packets = hax0r
    Backdoor password = s3cr3t
    Tag name = reptile
    Source port of magic packets = 666
    Reverse shell each X time = n

# 2. capture VM snapshot - Ubuntu 16.04.6 :: reptile injected

# 3. verify injection
whoami
> test
/reptile/reptile_cmd root
whoami
> root
```

Rootfoo Linux Rootkit:

```
# 1. download and install rootfoo rootkit
git clone https://github.com/rootfoo/rootkit
cd rootkit
make
sudo insmod rootkit.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: rootfoo injected

# 3. verify injection
dmesg | grep ROOTKIT hooked call; echo $?
> 0
```

Sutekh:

```
# 1. download and install sutekh
git clone https://github.com/pinkp4nther/sutekh
cd sutekh
make
gcc -o rs rootswitch.c
sudo insmod sutekh.ko

# 2. capture VM snapshot - Ubuntu 16.04.6 :: sutekh injected

# 3. verify injection
id | grep root; echo $?
> 1
./rs
id | grep root; echo $?
> 0
```