International Journal of Applied Information Systems (IJAIS) ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 10 - No. 8, April 2016 - www.ijais.org

Combating Kernel Rootkits on Linux Version 2.6 (Analysis of Rootkit Prevention, Detection and Correction)

T.J. Anande
University of Agriculture, Makurdi
Department of Electrical and Electronics Engineering
Benue State, Nigeria

T.K. Genger
University of Agriculture, Makurdi
Department of Electrical and Electronics Engineering
Benue State, Nigeria

J.U. Abasiene
Federal Polytechnic, Ede
ICT Directorate
Ede, Nigeria

ABSTRACT

Rootkits are a major security concern for smartphones today. They have always been around, though largely operational on desktops and PCs. On the mobile platform, their presence was not very popular until the advent of smartphones and advanced mobile devices. The rapid developments and trends recorded on smartphones today make them highly vulnerable to rootkit attacks. Smartphone operating systems now come highly sophisticated and packaged with advanced functionality to keep record of users' diary, sensitive personal and security details, among others. These features make them a prime choice for attacks from rootkit authors, who explore all available avenues to exploit and extract this information for malicious purposes. Cases of rootkit attacks have continued to increase with more of such attacks targeted at popular smartphone operating systems like Android. In this research, we discuss rootkits, illustrating their operational architecture and operation with a design of a kernel rootkit for the Linux kernel 2.6. We explore possible measures to combat rootkits on the mobile platform, using Android as a case study

General Terms:

Rootkit, Android

Keywords

Linux, Smartphone, Kernel, Kernel Freeze, Backdoor

1. INTRODUCTION

Rootkits are traced back to the UNIX Systems where operating system tools (Kit) were used to stealthily gain undetected privileged access (Root) into the computer system. According to Sirag, H. et al and Kim, S et al, a rootkit is software that maintains privileged access in a target system while evading detection from the administrator by altering or modifying functions in the operating system [19][38]. Lacombe, E et al in [20] defines them as a set of modifications that enable an attacker fraudulently and continuously control an information system using a combination of various malware techniques. Even though they are not harmful or malicious, they enable attacks from other malware like viruses, worms or Trojans, who use rootkit techniques to gain access into and infect the target systems via backdoors [12][36]. Backdoors are secret paths into secured systems - operating systems.

Panda Security [35] classifies rootkits as persistent or non-persistent; Persistent rootkits replicates valid registry files and are automatically activated each time the operating system starts up while the non-persistent rootkit is disabled once the operating system is restarted. According to Shetty, P, rootkits thoroughly examine their target's system as soon as they gain access. While examining the system, they collect all relevant information about the system to enable it identify possible vulnerabilities. As soon as weak points are identified, the attacker strategizes and attacks while concealing its tracks and presence in the system [36]. This essentially highlights its operational stages prior to and during an attack.

Rootkits are primarily deployed at the User Mode or the Kernel Mode. Other types include Memory based rootkits, Firmware rootkits, Boot rootkits and Hypervisor rootkits [19][23].

Kernel-Level Rootkits:. Presumably more dangerous and difficult to detect, this kind of rootkit attacks the nucleus of the OS (as shown in Figure 1). As soon as it gains access into the system kernel, it modifies and replaces kernel data structures. This rootkit type goes as far as hooking kernel Application Program Interfaces (APIs) using various hooking methods including System Service Descriptor Table (SSDT) Hooking, Interrupt Descriptor Table (IDT) Hooking, Direct Kernel Object Manipulation (DKOM), Inline Function Hooking (IFH) and Jump Template Technique (JTT) [19].

Kernel rootkits use SSDT Hooking to access and modify memory address functions and running processes through system despatch and parameter tables while it remains undetected. IDT Hooking enables rootkits obtain and manipulate software or hardware interrupts. This empowers it to control hardware and software responses to user commands. DKOM, unlike most hooking methods, is not based on execution codes. Attackers use DKOM to directly modify kernel-level data. With IFH, rootkits gain control over a specific function in the system memory by inserting malicious codes called unconditional jump in the body of the function [19]. The JTT is used when multiple functions are targeted; rootkit functions are initially executed followed by the original function code while the default function result is returned.

User-Level Rootkits. These run alongside basic application programs. They modify and replace APIs, registry values, system binaries and kernel utilities. User-level rootkit basically hook and conceal itself in APIs by substituting and inserting its code into the API's process address space [14] [36]. They are not as dangerous as the Kernel-level rootkits as they are not able to access and manipulate kernel structure but only manipulate APIs and user-level programs (as shown in Figure 1). They use Dynamic-Link Library (DLL) injection technique to insert their code into other processes which enable them execute inside, and spoof

International Journal of Applied Information Systems (IJAIS) ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 10 - No. 8, April 2016 - www.ijais.org

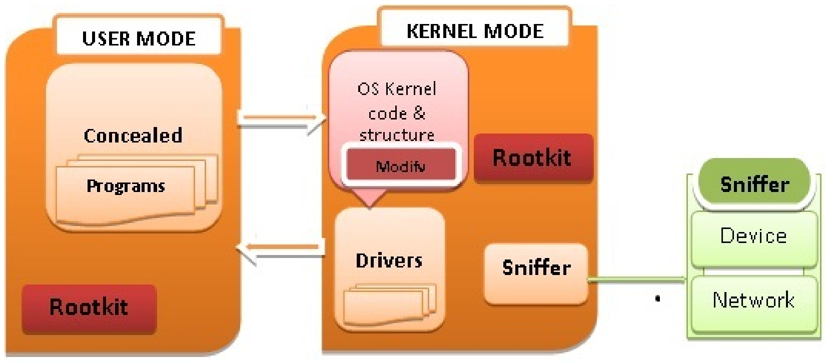

Fig. 1. Rootkit's position in a compromised system.

target processes [19]. DLL injection can be deployed in three ways; via registry, hooking function or remote thread. The other method is to overwrite the target application memory provided it has the required privileges [19].

1.1 Rootkit on the Linux Kernel

As earlier mentioned kernel rootkits are known to attack and subvert the core of the OS. According to [1], these rootkits are the most malicious type and could easily evade detection as they run alongside core kernel processes giving them the ability to subvert syscalls and generate false detection results for any anti-rootkit tool that is run on the system. Now most kernels are similar though not entirely the same. The Linux kernel (our subject kernel), like every other system kernel, serves as an intermediary that facilitates interaction between the user mode and the hardware. This interaction is made possible through system libraries (Native libraries as in the case of Android) via syscalls and functions [1]. Linux kernel rootkits run in the core layer of the OS generally referred to as ring 0 (where they have access to the system's highest privilege), modify the Loadable Kernel Module (LKM), and target these syscalls by subverting them.

1.2 Kernel Rootkit Analysis

In this section, we take a detailed analysis of a Linux Kernel Rootkit architecture stack and module stages. We also consider the design principle and processes. This section is based on an experiment on kernel-mode rootkit on the Linux-based system conducted by Lacombe, E, Raynal, F and Nicimette, V [20]. Other researchers have also experimented on kernel rootkits, but we use because it particularly relates to and expounds our subject Linux kernel rootkits.

1.3 A Functional Rootkit Architecture

As earlier stated, the primary goal of a rootkit is to stealthily compromise a system's defence system by intruding into the system, evading detection, and creating backdoors for continuous attacks from other malware. It is also important to note that these backdoors are also relevant to ensure constant communication with the rootkit authors (that is, if the rootkits are not local). Rootkit architecture is engineered in modules as shown in(Figure 2)

Module 1. This module implements the first and primary stage of the Linux kernel rootkit. Also called the Injector, it serves as the rootkit activator and injects the rootkit codes in the kernel modules. Whatever the purpose of the attacker is or vulnerabilities the system has the injector or activator must be implemented for the attacker to gain access into the system. Once the codes are injected, the kernel modules are compromised, and the rootkit immediately re-structures the kernel system to suit its op-

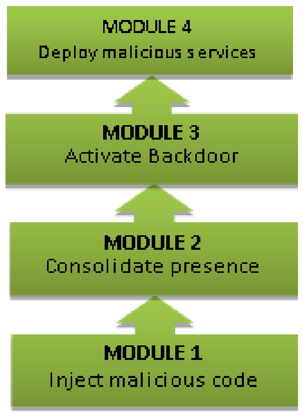

Fig. 2. The Functional Rootkit Stack.

erations. Different injection techniques can be employed depending on the kernel version; our focus is on Linux Kernel version 2.6. One injection method for this kernel is to dynamically include malicious kernel modules to the Linux kernel [41]; this is possible hence Linux kernel module (LKM) support is active. Alternatively, the */dev/kmem* (a device interface driver or character device used to access kernel memory via virtual addressing) is used to access, compromise and subvert the kernel [22]. Another approach is to bye-pass the operating system (or processor) and directly accesses the system's memory management. Injecting the code through */dev/kmem* could disable the LKM as well as disable execution of core system programs.

Module 2. This is the Rootkit Protection Module. As shown in Figure 3, it is responsible for ensuring the
rootkit is robust, resistant, persistent and sturdy. Basically, the attacker employs and combines different strategies to achieve this. Whatever strategy is employed, the attacker must ensure the rootkit is invincible and actively running along with the system. Persistent rootkits would inject part or all of their code in the kernel and remain active even when the system is restarted. One way these kinds of rootkits defend themselves against anti-rootkit measures (especially when detected) is by giving the user or administrator the impression that removing them could cause more harm than good. By disguising their operations (that is, processes, files and network sockets), they would manipulate, modify and even erase event log files, hook system call table or virtual file system, or other system functions.

Module 3. As soon as rootkits establish their presence in a system, they would proceed to create Backdoors to enable unlimited and sustained access to and control over the system by the attacker. Figure 4 shows that with Backdoors, the attacker would

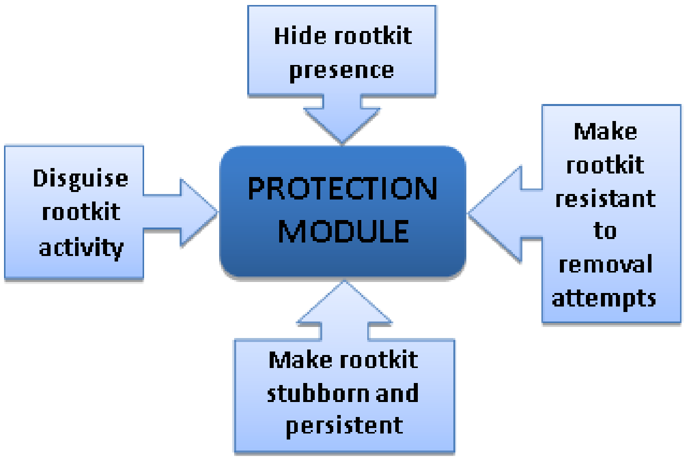
Hide rootkit presence
Disguise rootkit activity
PROTECTION MODULE
Make rootkit resistant to removal attempts
Make rootkit stubborn and persistent

Fig. 3. Various strategies employed at the rootkit protection module.

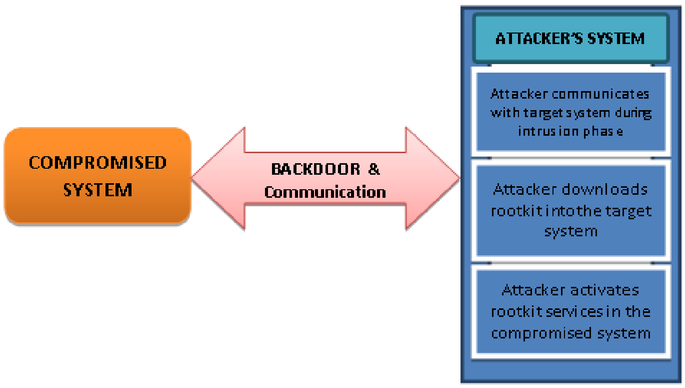
ATTACKER'S SYSTEM
Attacker communicates with target system during intrusion phase
Attacker downloads rootkit into the target system
Attacker activates rootkit services in the compromised system
COMPROMISED SYSTEM
BACKDOOR & Communication

Fig. 4. Communication process before and during rootkit deployment.

continue to exploit and extract sensitive information from the user. This Backdoor, activated by its actuator (a mechanism that automatically activates it), serves as the interacting channel that connects the attacker and the rootkit operation in the compromised system. The interactive channel communication connects the attacker's device or system to the compromised system at one end, and provides a pathway for the rootkit to execute instructions from the attacker on the compromised system at the other end.

Module 4. Once the Backdoor mechanism is set and running, the rootkit proceeds to deliver its services in the compromised system. Services (see Figure 5) such as spying on conversations, keylogger (surveillance mechanism that registers e-mails, keystrokes, passwords, instant messages, and contacts) [45], and other such activities are classified here as passive services while active rootkit services include DoS, terminating application or system processes, corrupting file systems, intruding other systems. The attacker would usually use passive services to steal information from the compromised system, and employ active services to disrupt the system or kernel operations.

2. THE KERNEL ROOTKIT DESIGN OBJECTIVE

This rootkit is design for deployment on Linux Kernel version 2.6 (based on x86 architecture). This section details the architectural modules for the design, starting from the subversive method employed followed by the protection mechanism, creating the backdoor, and finally deploying the rootkit services. The objective is to subvert a process (thread) - system call 0, and ensure all modifications made are hidden from other active processes [20]. System call (syscall) is a mechanism used by user-land processes to request OS managed services such as process management, network, memory and storage, from the kernel [11]. System call 0 (syscall 0) runs in the x86 architecture's highest privilege level, and is required by the kernel when restarting interrupted system calls. These interrupted system calls would usually store their address in their process descriptor; accessing and modifying this temporarily stored address is the main goal of this kernel rootkit.

2.1 The Kernel Rootkit Design

[21] creates two separate codes; the first code modifies syscall 0 such that it is able to call kernel functions, and the second code serves as a backup to enable [20] deploy functions other than the available kernel functions. These codes could be deployed locally by modifying syscall 0 sequences and launching them from within the compromised system. [20] could alternatively deploy the codes remotely by modifying the syscall 0 sequences which then connive with the kernel backdoor to execute commands from the [20]'s system onto the compromised system.
Code Injection: [20] initiates installation by accessing and opening the */dev/kmem*, and proceeds to locate get-zeroed-page primitive address (obtained from a physical memory page) using */dev/kmem's* pattern matching mechanism. get-zeroed-page primitive is part of kernel memory allocator's very low level call. The next step is to locate the process stack and descriptor. Based on kernel version (Linux 2.6), [20] proceeds to identify the value of the esp stack pointer to be able to get the process' kernel stack. This is done in kernel mode, during the process execution. [20] also accesses the descriptor's reference within the thread-

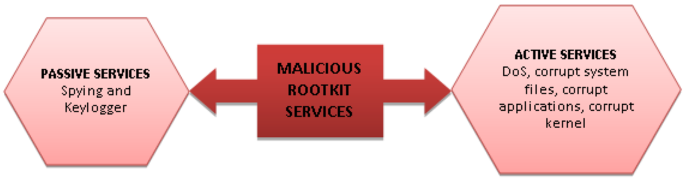
PASSIVE SERVICES
Spying and Keylogger
MALICIOUS ROOTKIT SERVICES
ACTIVE SERVICES
DoS, corrupt system files, corrupt applications, corrupt kernel

Fig. 5. Malicious rootkit's Active and Passive services.

info structure at the lower part of the kernel stack (see Figure 6). Thread-info is one of Linux's two data structures used for each (individual) process, and the esp is the stack pointer register; it is configured with the eip (that is, the instruction pointer register) to compile loadable set values deployed for registers during system initialisation and switch system processors to kernel mode (otherwise referred to as ring 0) [8].
Now, as soon as the processors switch to ring 0, the esp value of the particular scheduler stored within the tss-struct is loaded with the esp. At the completion of the sysenter execution, esp register loads revealing the stack address of the executed process; hence it loads with the structure's address. Note that sysenter is executed from user space. Sysenter is an upgraded system call mechanism that optimizes and boosts processor performance when switching to ring 0 by determining eip, esp and other such registers based on OS set values [11]. The tss-struct is a structure reference type in x86 architecture used to build up the Task State Segment (TSS). After locating the process kernel stack and process descriptor, [20] proceeds to inject the code (code 1) to subvert syscall 0, by replacing the called function's address with an address from code 1 (the injected code). This allows the injected code run in kernel mode and obtains its memory page address allocated by the kernel. [20], further, uses */dev/kmem* to inject a second code (code 2) in the memory page to enable any kernel function execute from user space; this replaces the allocated address (used by syscall 0 and referencing the injected code inside the kernel) with the address of code 2 (newly injected code). By these modifications, syscall 0 semantic is altered such that whenever syscall 0 is called, code 1 address (rather than the default function's address) runs in ring 0 while code 2 collects whatever command or parameters transmitted between the process kernel's stack and syscall 0. Once these parameters are met, the required function can be called, paving the way for [20] to infiltrate the compromised system's user space with the rootkit's schema.

Concealing The Kernel Rootkit. Two methods - online or offline, can be employ to conceal this rootkit; however, [20] focuses on online (while the system is connected and running) techniques in this work as the subject systems (smartphones) are mostly online. Two approaches can be used for this method; subverting virtual memory allocation (VMALLOC) which is enforced in Linux based on memory management unit (MMU), or alternatively, to modify MMU Control Bits. [20] dwells more on subverting VMALLOC as the essence is to basically demonstrate how this rootkit can be concealed in the compromised system.
Subverting VMALLOC: Linux uses VMALLOC as a dynamic memory allocation function for virtual memory allocation [24]. [20], by this approach, enter the kernel space, and allocate and loads malicious code into a memory page using VMALLOC non-contiguous memory allocator. Now, MMU holds process addresses that come with Page Global Directory (PGD). A mechanism called Context Switching individually loads these addresses during every context switch process, thereby separating processes such that they do not interact with each other. X86 architecture basically reserves about 4GB interval in ring 0 for kernel linear address space which are linked to the same physical addresses, irrespective of the process; note that there is a unique address space for every process. The memory page that holds the malicious code also has a linear address, and it is stored in an area allocated for VMALLOC address space. Between adjacent or immediate pages of the linear address space and physical pages in this area, linear addresses must not match a corresponding physical address unlike the rest of the address space in the kernel. Thus, [20] uses VMALLOC to book two physical memory pages. One of the pages is loaded with malicious code while the other is left empty. We will assign lm as linear address for the malicious page, and pm as its physical address. We will also assign le as linear address for the empty page, and pe as its physical address. Now, [20] triggers this process to allocate the memory pages by accessing the primary PGD, searches for lm so as to acquire pm. The same process is repeated for le and pe. As soon as pm and pe are acquired, the process switches them (pm with pe) in the primary PGD entry, and modifies its own PGD to synchronise with the updated primary PGD (see Figure 7)This gives the process exclusive and uninterrupted access to the malicious page.
Activating The Backdoor: Even though this rootkit is a kernel rootkit, [20] does not necessarily require root privileges to execute it as it can be implemented from user space. Two backdoors can be implemented for the rootkit; one for local deployment and the other for remote deployment. Local Backdoor: [20] actuates the backdoor by disconnecting and concealing the kernel thread, so it becomes hidden in */procfilesystem's* thread list, and further remain undetected by any system activity tool. [20] uses signals in user to interact with the kernel thread, which means the thread is referenced in a hash table - this hash table is used during inter-process communication; hence its concealment is only partial. Using a set signal handler, the thread is able to respond to any signal transmitted from the user space by locating its corresponding transmitter within the list of user processes, and updating system call 0's address to code 2 address. Alternatively, [20] can block communication between the thread and the hash table and set it to non-permanent sleep mode in the processes lists. This way, user space is not able to communicate with the thread anymore, and because the sleep mode is also periodic, the thread occasionally scans and locates the corresponding process descriptor among a set of process descriptors, then updates system call 0's address with code 2's address, and loads code 2 into the process. Remote Backdoor: Much of the conniving and concealing process is implemented within the compromised system. This does not imply that [20] cannot deploy a backdoor that facilitates remote access to the system; this mechanism, however, may require some form of authentication. The rootkit logic is remotely processed on [20]'s system and deployed on the compromised system via syscall-proxy (a command that enables system call execution over the network [42]); syscall-proxy is loaded in a compromised local process. Alternatively, [20] can employ a mechanism called mobile parasitism. This parasitic algorithm is stealthily executed in the compromised system via kernel processes or threads. [20] loads malicious code into the kernel memory of the compromised system. The loaded code modifies the memory to make the thread execute this code, and infest the kernel threads or processes with some parasitic loop. This mechanism is further developed, but we do not extend our research

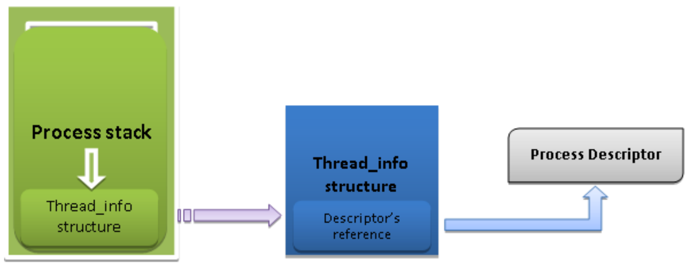

Fig. 6. Process kernel stack and process descriptor location.

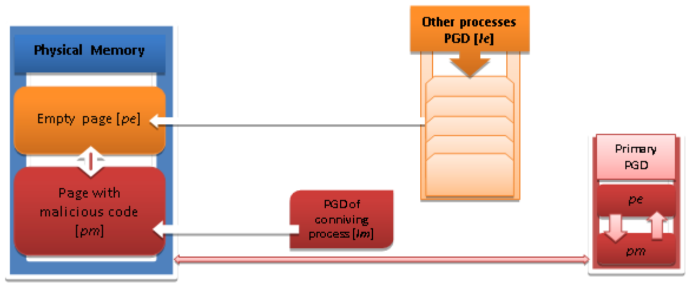

Fig. 7. VMALLOC subversion process in the MMU.

beyond this point. Deploying Services: With the aid of the backdoor, [20] is empowered by the rootkit to infiltrate the compromised system in so many ways without detection. Disconnecting processes and threads from core kernel structures as illustrated in the local backdoor limits system kernel - thread interaction, which further conceals [20]'s malicious activity in the kernel of the compromised system. With mobile parasitism mechanism, [20] is able to remotely maintain stealthy malicious deployment using the loop algorithm thereby extending the infestation process on kernel threads.

2.2 Limitation of this Design

As earlier stated, this design is aimed at basically demonstrating one of the many ways a kernel rootkit can be conceived, deployed and concealed. Other authors would implement their design with more sophistication, depending on what their motive and objective is. This design, though operational, does not come without limitations. The kinds of limitations in every design also depend largely on the sophistication of the design.

One way this rootkit presence can be compromised is through monitoring the user space - kernel process interaction. System call 0 is a core kernel process, but [20] calls it from user space which is not normal system behaviour. As long as the system administrator can trace this behaviour, there is a chance that this rootkit will be compromised.

Linux kernel 2.6 comes in different versions; this design may not be compatible with the kernel structure of some (especially newer or updated) versions.

The technique employed to subvert VMALLOC (using its non-contiguous memory allocator) can be time consuming and equally strain the processor.

These backdoors can only work when [20] is actively in the system or remains connected to the system.

Corrupting multiple kernel processes or threads (as done by mobile parasitism) risks exposing this rootkit presence.

The Linux kernel rootkit experiment demonstrated here is an indication that Android (which is based on Linux kernel) could also be vulnerable to this rootkit. As mentioned earlier, the design is particularly deployed on Linux kernel V2.6 - same kernel with Android. In the next section, we discuss possible ways of preventing or mitigating these rootkits.

3. ROOTKIT PREVENTION, DETECTION AND CORRECTION

Rootkits, in all their complexity and sophistication, do not exist without a fight from operating system owners and anti-malware producers. As the trend shifts from conventional to more sophisticated rootkit operation techniques, mobile operating system manufacturers also seek out more improved and advanced ways of securing their operating systems from imminent and/or potential attacks. OHA's Android, one of the leading mobile operating systems in use today, is constantly reviewed and updated to beat the increasing security challenges faced today [6][17][9]. Anti-malware producers like McAfee, Kaspersky and Anti-Virus Guard (AVG), among others, have continued to explore all possible system vulnerabilities and rootkit attack techniques with a view to update their databases as well as offer maximum protection to system (especially smartphones) users via their anti-malware solutions. While absolute protection from attacks is never guaranteed, anti-malware software in use today are also empowered to mitigate the harm rootkits can cause on a compromised system. Despite these efforts, attackers are still able to find vulnerabilities as a result of the increasing complexity that comes with every operating system upgrade. The operational techniques of rootkits make them a very big challenge on compromised systems because they become very difficult to detect and mitigate or remove. In this section, we consider possible ways of preventing rootkit attacks, as well as techniques of detecting and correcting attacks on a compromised system - with emphasis on Android.

3.1 Prevention

According to Jack Wallen, prevention is possibly the best protection against rootkits [43]. Schultz and Ray in [34] maintain that preventive measures ensure optimal protection for the system as a compromised system will be subjected through detective and corrective measures; which do not always restore the system's integrity. We discuss various possible methods to ensure maximum prevention of rootkit attacks. These methods are grouped into Manufacturer and Developer roles, User roles, and Third Party roles.

3.2 Developer Roles

Developer here include OS developers, developers of Applications (third party applications not included), and developers of security infrastructure (such as Google). These are responsible in ensuring the operating system is as secured as possible before it is released into the market. From the technical point of view, Android Team has continued to follow-up on their product with constant improvements. Much, however, remains to be done on their governing policies.

Ownership And Control. Rootkits largely rely on the user's action to activate themselves on a smartphone. Normally, they would attach themselves to applications (usually applications that run with major system privileges) and prompt the unsuspecting user for activation. As soon as the user enables or installs the application, the rootkit becomes active in the device. Android operates an open policy which gives the end-user a lot of privileges. This enables the user make major decisions and changes on their system, including what application to activate. A lot of these decisions and changes, unknown to the user, have security implications. With such control by the user, the Android Team can only exert absolute control over their product prior to it getting into the hands of the end-user. Once it gets to the user, Android Team's control becomes subject to the user's consent. Even though, this seems to be a deliberate attempt by Google to ensure Android is more user friendly, it has not really guaranteed the user's safety. Google sometimes uses Remote Kill Switch to remotely revoke and disable or uninstall malicious applications on android powered devices [3], this technique may have little or no effect as the rootkit would have already achieved its intended objective on the smartphone. We therefore propose that by reducing some of these privileges, Google will be able to gain more control over their product even when it is out in the market, and reduce the increasing incidences of rootkit attacks. This can be achieved by withdrawing most of the privileges available to users and allowing users only have access to privileges that have less security implications.

User Awareness And Orientation. According to Toyssy and Helenius [40], malwares (including rootkits) largely spread via social engineering. The user accepts and installs malwares disguised as games or other applications without knowing it. Toyssy and Helenius suggest that educating users on the risks associated with installing suspicious applications would reduce the spread of mobile malware. This, however, is only being researched and not yet implemented. On the technical side, Android Security Team and Google's Information Security Engineering Team are each saddled with individual responsibilities to monitor Android for possible vulnerabilities before and when it is already released into the market. While this system helps OHA maintain a close check on attacks, it still does not prevent attacks from happening. This strategy from OHA does not really prevent rootkits from attacking Android smartphones as attacks still largely depend on the user's permission [40]. OHA can create special programmes targeted at educating their users on the dangers of rootkits, ways they use in attacking, and what the user needs to do to avoid being vulnerable, both organisations would achieve more in securing the user as well as their products. This programme would inform users on safe surfing habits, common techniques employed by attackers, common rootkit behaviour in a compromised system. This approach does not guarantee an end to rootkit attacks; however it can help the user protect guard against imminent attacks. Unfortunately, the average user wants to explore the capabilities offered by the smartphone with little attention to the security implications associated with these features. Thus, both companies may want to explore the best approach to get users to subscribe to this programme.

Admission Control. As earlier mentioned, rootkits conceal themselves in applications and attack as soon as the user activates the application on their device. While it is difficult for such applications to go through Android Market Place without being detected, it does not prevent the user from having access to them. Installing third party applications on smartphones can be monitored and checked by ensuring that all third party applications are subjected through a standard vetting system that checks for digital signature conformity and application source [3]. This system could be incorporated as a function into the operating system and/or enforced during the process of download.

Kernel Freeze. Deep Freeze is created by Faronics and is presently available for Microsoft, Mac and Linux powered computer systems. It is designed to enable systems revert to pristine configuration after restart, thus securing the core and configuration files of operating systems. This means whatever changes that are made on the system including installations, whether done in error or intentionally, is only temporal [7].
We propose that a similar operational technique called Kernel Freeze be incorporated into mobile OS kernel; especially for Android kernel. By our proposal, Kernel Freeze (see Figure 1.7) will be a function in the kernel, and will require the operating system to restart after every new installation on the system. After the restart, whatever changes that were made in the kernel will revert to the original configured state. This way, both the system kernel and the application sandbox will remain immune to changes made during installations; any kernel rootkit installed during application installation will be erased as soon as the Kernel Freeze function restarts the system. This function will only work at the kernel level as implementing it at the user level may result to loss of user's personal data and information.

3.3 User Roles

Like the operating system manufacturer and application developers, the users also have responsibilities in preventing rootkit attacks. The user, according to [40], largely determines the success of rootkit attacks as he or she must consent either consciously or unconsciously. While the manufacturer is expected to educate the user on the dangers of and ways to avoid rootkits, the user is required to apply precautionary measures. Social Engineering: Rouse, M [30] defines social engineering as non-technical means of invasion that largely depend on social interaction and is based on deception that results in compromising security standards. Through social media networks, smartphone users are deceived into exposing their private and confidential information; the attacker takes advantage of their natural vulnerabilities to gain their trust and exploit them. Heary, J. in [15] identified popular social engineering techniques employed by attackers to include familiarity with potential victims, hostile attitude to evade interrogation, facts about the target to capture their attention, reading and exploiting the target's body language. While a lot of security experts recommend social awareness and training on social engineering techniques for users, we also suggest personal discipline and discretion towards personal privacy at all times. Users need to realise the value and worth of the kind of information they reveal via smartphones, and the consequences of having such information compromised. This training could focus on such areas as those highlighted in the next topic (Installations). The challenge with social engineering, however, is more due to human nature rather than a technical issue; which makes it quite difficult to tackle as human beings tend to react differently based on their intuition and instincts. Social engineering definitely cannot be stopped, but with a better understanding, users would apply more discretion towards the kind of information they reveal over their smartphones and whom they reveal

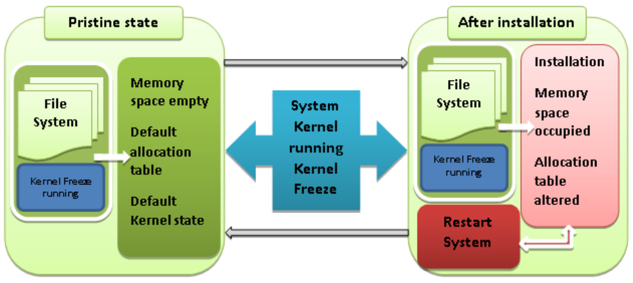

Fig. 8. Kernel Freeze revert system kernel to the pristine state after restart.

the information to. Installations: Android users are more secured when they download applications from the Android market Place than using external sources. This is because the user may not be able to identify trusted or secured sites. During installations, users may also need to be careful about the permission requests they approve as some have security implications [28]. Users need adequate knowledge about any third-party application before installing it on their smartphone. It is the user's responsibility to have a good anti-rootkit solution running on their smartphone. Popular anti-rootkit producers today include Kaspersky, AVG and Panda, among others.

3.4 Third Party Roles

There is a need for a controlled third-party application market that acts as a vendor repository where third party applications (including extensions) are subjected through a vetting process before they are admitted [3]. This means smartphone application vendors, especially for Android, will be required to adhere to secure coding principles [27] and privacy policies [28]. OHA control policy does not include exerting total control over the application market as it may affect Android's scalability, this comes at the cost of user security as the policy relies on the end-user to perform the vetting through his/her interaction with third-party developers.

3.5 Detection and Correction

Butler and Hoglund in [16] observed that the same prevention mechanisms employed against rootkits are also available to rootkit developers and so, blocking an attack does not necessarily mean completely preventing intrusions as in the end, it depends on who (between the rootkit and the prevention mechanism employed) gains access to the system first. And because today's rootkits are constantly increasing in sophistication, detection and correction have become somewhat an uphill task. There is no known best method for correcting a rootkit attack because, somehow, every method has a form of weakness that rootkit authors always seem to find their way around. Kernel level attacks, for instance, may best be corrected by completely re-installing the smartphone's operating system; but this may also render the device useless especially when not properly done. Several approaches have been employed (and more still been researched and proposed) to detect and remove rootkits from compromised systems. Butler and Hoglund in [16] further group these approaches as Rootkit Presence-based Detection and Rootkit Behaviour-based Detection. We will discuss them in the following sections.

3.6 Presence-Based Detection

This approach involves searching the operating system for rootkit images as earlier rootkits based their attacks on the operating system files. With rootkits advancing their attacks from file systems to hardware such as BIOS and system memory, there was a need to upgrade from just offering software-based detection to including hardware-based detection as well [16]. Presence-based detection is employed when an intrusion is confirmed. Such methods as File Integrity Checks, Memory/Kernel Debugging, and offline checks are either combined or used individually in identifying and isolating the rootkit.

Offline Checks. Butler and Hoglund believe that offline checks produce the best results in detecting rootkits on a compromised system [16]. The ability of a rootkit to remain immune and undetected is only possible when the host (operating system) is running; thus, shutting the operating system down is the best way to compromise the rootkit's immunity [26]. The compromised system is switched off and re-booted via an alternative but trusted source such that the 'victim system' is accessed as a secondary system or inactive device (slave). This way, the rootkit is rendered inactive as the operating system where it resides is no longer running, and the operating system or file system is then examined to determine the extent of the damage as well as possible restoration. This method may not be suitable for smartphones as users always need them on; shutting them down could result to denial of service because their ability to communicate at that moment will be grounded. Performing this diagnosis on them while they are still running is possible but may result in compromising the trusted system as the rootkit will still be active and so, could evade detection, and attack and compromise the system. More so, smartphones are usually connected to such systems (in this case, computer systems) via USB ports which limits access to some of the phone's configuration files especially the kernel's dynamic file system [4]. While smartphone users may be more concerned with access to both service and communication, security should be a priority as a compromised system cannot guarantee secured access. The success of this method, however, depends on the type of rootkit attack; it may not be a viable solution for kernel level rootkits. Memory/Kernel Debugging: This method involves diagnosing the system memory or kernel memory (depending on the attack level) through system memory or kernel memory dumps. Memory Dump refers to the visual display and backup of memory contents to enable diagnosis in the event of system or application failure [18]. Forensic analysis, testing and debugging is performed on the system files, or kernel to reveal memory data and code (including passwords), and other activities including any malicious existence in the memory [25]. System memory dump is either complete include the entire physical memory contents, or mini includes loaded device drivers, current processes, parameters, among others. Kernel memory dump only contains read/write pages in the kernel-mode [13]. Generally, memory dump enables the system administrator access vital information about the system and installed programs in a previously pristine state [18]. This enables specialists diagnose and identify existing rootkits. Due to the complexity of this method, it is best performed offline and by specialists; this may also result in denial of service when applied on smartphones hence this method best works when the system is offline. However, [37] reveal that another approach to detect such rootkit (especially if they reside in the hardware) while the system is still running, and not having to reboot the system afterwards, is by bypassing the upper layers including the kernel, and examining the underlying hardware itself.

File Integrity Checks. This method requires a Personal Computer (PC) and is time consuming. Basically, a rootkit detection and file synchronization software is installed on the PC and initiated to access and carry out file integrity checks on connected smartphones by navigating through and manipulating the phone's file system [10]. This allows the software on the PC to download and scan the files on the smartphone for malicious codes, and isolate compromised files which are either restored or deleted (depending on the level of infection). Compromised files are identified by comparing hash functions form codes in their pristine state created and stored on the PC with the hashes from the downloaded files from the smartphones [10]. Both hashes are compared to determine if any modifications have been made on the file downloaded from the phone. This process may consume battery life due to the amount of time involved in downloading and scanning all files, however, this can be minimized if the software only downloads and scans only modified files. This method may not be very effective against a kernel-level rootkit as such rootkit has the tendency to disguise itself as the operating system thereby manipulating hash queries to evade any detection.

3.7 Behavior-Based Detection

This is a cautionary approach employed to monitor system performance against rootkit-like activity in smartphones. It is very powerful as it exposes the rootkit in its disguised state as well as its attack objective [16]. Such methods as Heuristics Detection, Signature-Based Detection and Analysis Detection are behaviour-based.

Signature-Based Detection. This is a major method employed by most anti-rootkit solutions in detecting rootkit behaviour on compromised systems. Most anti-malware solution vendors keep an updated database of coded signatures of known malware, including rootkits. During a system scan, the anti-rootkit software cross-indexes its database signatures with those of the system files to check for similarities or regular patterns [30]. Files whose signature codes match are marked as suspected or compromised files, and are either quarantined or disinfect or deleted. The challenge with this method is the new trend where rootkit authors now use polymorphic and metamorphic codes to evade such detection [5]. Polymorphic codes come in two parts; the original algorithm remains unchanged at the background while the other code part mutates whenever it runs. Metamorphic codes, on the other hand, dynamically change during every iteration making them appear as different programs running while it is actually one program appearing as many. Both codes, especially metamorphic codes, have increased the complexity of emerging rootkit thereby making it more difficult for them to be easily detected by anti-malware solutions. To combat such complexity, Rouse in [30] recommends the use of Heuristic Analysis - an advanced method that involves Virtual Systems and Generic Decryption. Graph isomorphic test, another detection method applied for polymorphic codes, is able to detect polymorphic code structures even when executed at different times and levels (including system kernel) especially because the original algorithm holding its structure does not change [29].

Heuristic Analysis. Signature-based detection is only effective against malware whose signature codes match those in the anti-malware solution database [33]. Due to this weakness with signature-based detection, anti-malware authors added a proactive feature commonly referred to as Heuristic Analysis technique which enables anti-malware programs detect previously unknown malware threats (even if it is not found in their databases). Heuristic Analysis in use today largely depend on rule-based mechanisms - a system that compares extracted file rules with malicious code rules and triggers when it identifies matched rules [39]. However, an older weight-based mechanism (an approach that calculates and measures detected weight against a standard threshold) is sometimes used or combined for better results [32][31]. This technique can be implemented via Virtual Systems and Generic Decryption. We discuss these implementation methods below.
Virtual Systems: The use of virtual machine based rootkit detection is gradually gaining prominence especially as it promises efficient rootkit detection for smartphones. This method offers an isolated environment running concurrently with the authentic operating system on the same smartphone. Anti-rootkit solutions run on the virtual environment and, occasionally access and scan/analyse the smartphone's memory for malicious activity and file structure integrity. This process is quite intensive and drains battery life which is why solutions like Gibraltar rootkit detection system that frequently collects and scans kernel memory snapshots for advanced rootkits via kernel data structure modifications can be re-configured to reduce its scan frequency; this, of course, does not guarantee better security even though battery life is improved [2][4]. Generic Decryption: Generic decryption is generally employed for polymorphic and/or metamorphic malware. This process is deployed via a scanner that uses generic encryption methods to monitor and analyse system files in an isolated and controlled virtual environment [44]. This way, any malicious program infecting a file is exposed to the scanner as it decrypts during the file execution. Earlier versions of generic decryption had speed limitations; however, the reviewed version is heuristic-based. Rather than scan and analyse all files, it checks for inconsistencies and variations in programs and file behaviour; programs or files exhibiting such behaviour are isolated, monitored and analysed in the controlled virtual environment until the decrypt exposing the malware [44].

4. CONCLUSION

We explored the evolution and operational techniques of mobile rootkits. At the user space, mobile rootkits could be easily contained due to the application processes isolation mechanism implemented on Android. On the other hand, kernel rootkits are presumably very dangerous and destructive. Because they are able to manipulate and change the kernel structure, their presence in any system compromises the core of the operating system. We further demonstrated the architecture and design of a mobile kernel rootkit designed for deployment on Linux 2.6 kernel. In the design (extracted from [20]), we showed how, with two separate codes, [20] could inject and run a rootkit in the kernel mode from the user space. By subverting the VMALLOC, the rootkit was able to manipulate the system?s memory management addressing scheme to enable it conceal itself. We also showed how [20] created local and remote backdoors to facilitate sustained connection with [20]'s system (whether locally or remotely), as well as deployment of the rootkit services on the compromised system. Though operational, we identified limitations in this design which could enable the system administrator or users detect and compromise the rootkit presence. Among such limitations was the fact that the rootkit ran from the user space, and constraint with different kernel versions. We also discussed protective measures against these rootkits for smartphone users, highlighting roles for developers (both OS and Application developers), Operator Equipment Manufacturers and users in the fight against rootkits. Reducing privileges available to users, enforcing user awareness and orientation systems, guarding against social engineering methods, using anti-malware solutions, and personal discipline, are ways users can be secured from these attacks. Our proposed kernel freeze system, if implemented, would secure the OS kernel from any rootkit installation. Enforcing proper admission control over third-party developers, the Android market place would also limit infiltration of rootkit deployment from malicious developers. Preventing a rootkit attack is always the most secure protective measure, but considering the increasing complexities that accompany emerging rootkits as relating to mobile OS upgrades, it is not always possible to prevent such attacks. Due to this fact, we explored detective measures for incidences of infections or as proactive steps against attacks. Popular measures like offline and file integrity checks, memory/kernel debugging, heuristic analysis, and signature-based detection have been employed in detecting rootkits on systems; though some are not very efficient on mobile platforms. Thus, the possibility of overcoming the rootkit menace is largely dependent on the ability of industry players, including developers and OEMs, to adhere to required operational standards and instituted security policies. There has to be a compromise on financial gain for a more secured environment where the end-user's safety and trust are the priority. This will not stop rootkit authors from releasing more rootkits though, but it will minimize the success rate of their attacks.

5. RECOMMENDATION

The rootkit demonstration in section 3, as earlier stated, is aimed at basically showing how it can stealthily attack and compromise a system's kernel. More complex rootkits would not only subvert the system call 0 from the user space, but directly run in the kernel space. This is more dangerous and difficult to detect. Thus, there is a greater need to secure the kernel (in this case, the Linux kernel) from any form of rootkit attack, whether locally or remotely. Kernel freeze mechanism will not only restore the kernel to its pristine state, but will also ensure that the core of the OS remains in its default mode. Attackers will definitely want to compromise the restore mechanism by attacking the freeze function. Thus, we recommend that further research be carried out on how to restrict access to the freeze mechanism from developers. During our investigation, we discovered that very little or no research focused on implementing this mechanism on the mobile kernel has been done. Thus, we also recommend that implementing the results on the Linux kernel be the focus of this research. This requires the inclusion of the freeze function in the kernel; either by OEMs or developers (especially Android developers). User awareness and orientation programmes that enlighten users on the consequences that accompany their control and decisions made on their smartphones will not only make them more conscious of their security needs, but will also help them apply these privileges wisely. As part of the device activation process, an interactive session (basically titbits) on the user's role in ensuring maximum security on the device can be included to run while the user activates the device.

6. REFERENCES

[1] Rootkit Analytics. Kernelland rootkits. Available at `http://www.rootkitanalytics.com/kernelland/linux-kernel-rootkit.php` (2013/12/09).

[2] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *Dependable and Secure Computing, IEEE Transactions on*, 8(5):670–684, 2011.

[3] David Barrera and Paul Van Oorschot. Secure software installation on smartphones. *IEEE Security & Privacy*, (3):42–48, 2010.

[4] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the eleventh workshop on mobile computing systems & applications*, pages 49–54. ACM, 2010.

[5] A. Bridgewater. What is signature based detection. Available at `http://www.blogs.avg.com/business/signature-based-detection/` (2013/09/30).

[6] Andriod Community. Security. Available at `http://www.source.android.com/security/index.html` (2016/02/20).

[7] Faronics Corporation. Faronics deep freeze enterprise: User guide. Available at `http://www.faronics.com/assets/DFE_Manual.pdf` (2013/09/12).

[8] P Bovet Daniel and Cesati Marco. Understanding the linux kernel. *Sebastopol, CA, US, OReilly*, pages 500–800, 2005.

[9] Nokia Developer. Windows phone platform security. Available at `http://www.developer.nokia.com/Community/` (2013/08/28).

[10] Bryan Dixon and Shivakant Mishra. On rootkit and malware detection in smartphones. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 162–163. IEEE, 2010.

[11] Manu Garg. Sysenter based system call mechanism in linux 2.6, 2006.

[12] Hoglund Greg and B James. Rootkits: subverting the windows kernel. *H. Greg, & B. James, Rootkit Detection*, pages 295–312, 2005.

[13] C.C. Hameed. Understanding crash dump files. Available at `http://www.blogs.technet.com/b/askperf/archive/2008/01/08/understanding-crash-dump-files.aspx` (2013/09/26).

[14] Jie Hao, Yu-Jie Hao, Zhi-Jian Ding, and Lin-Tao Song. A methodology to detect kernel level rootkits based on detecting hidden processes. In *Apperceiving Computing and Intelligence Analysis, 2008. ICACIA 2008. International Conference on*, pages 359–361. IEEE, 2008.

[15] J. Heary. Top 5 social engineering exploit techniques. Available at `http://www.pcworld.com/article/182180/top_5_social_engineering_exploit_techniques.html` (2013/09/13).

[16] Greg Hoglund and James Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[17] IDC. Idc press release. Available at `http://www.idc.com/getdoc.jsp?containerId=prUS24108913` (2013/08/01).

[18] C. Janssen. Memory dump. Available at `http://www.techopedia.com/definition/20663/memory-dump` (2013/09/26).

[19] Sungkwan Kim, Junyoung Park, Kyungroul Lee, Ilsun You, and Kangbin Yim. A brief survey on rootkit techniques in malicious codes. *Journal of Internet Services and Information Security*, 3(4):134–147, 2012.

[20] Eric Lacombe, Frédéric Raynal, and Vincent Nicomette. Rootkit modeling and experiments under linux. *Journal in Computer Virology*, 4(2):137–157, 2008.

[21] H. Lang. Freebsd kernel rootkit design howtos - 4 - kernel and user space transitions. Available at `http://www.old.hailang.me/2012/06/10/freebsd-kernel-rootkit-design-howtos---4---kernel-and-` (2013/08/27).

[22] Anthony Lineberry. Malicious code injection via/dev/mem. *Black Hat Europe*, page 11, 2009.

[23] B. Martin. Types of rootkit viruses preventive measures. Available at `http://www.dailytipsndtricks.blogspot.com/2013/02/types-of-rootkit-viruses-preventive.html` (2013/07/15).

[24] Naveen. Embedded linux. Available at http://www.naveengopala-embeddedlinux.blogspot.co.uk/2012/01/linux-kernel-programmingmemory.html (2013/11/13).

[25] Nixcraft. Top 8 tools for search memory under linux/unix [forensics analysis]. Available at http://www.cyberciti.biz/programming/linux-memory-forensics-analysis-tools/ (2013/09/26).

[26] PC Plus. How to discover hidden rootkits. Available at http://www.techradar.com/news/computing/pc/how-to-discover-hidden-rootkits-1095174 (2013/09/25).

[27] OWASP Mobile Security Project. Android. Available at http://www.owasp.org/index.php/ (2013/09/17).

[28] Srikanth Ramu. Mobile malware evolution, detection and defense. *EECE 571B, Term Survey Paper*, 2012.

[29] Rizwan Rehman, GC Hazarika, and Gunadeep Chetia. Malware threats and mitigation strategies: A survey. *Journal of Theoretical and Applied Information Technology*, 29(2):69–73, 2011.

[30] M. Rouse. Social engineering. Available at http://www.searchsecurity.techtarget.com/definition/social-engineering (2013/09/23).

[31] M. Rouse. Trojan horse. Available at http://www.searchsecurity.techtarget.com/definition/Trojan-horse (2013/10/23).

[32] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub. A survey on malware and malware detection systems. *International Journal of Computer Applications*, 67(16), 2013.

[33] Markus Schmall. Heuristic techniques in av solutions: An overview. *SecurityFocus. com, http://www. securityfocus. com/infocus/1542,(Feb. 2002)*, 2002.

[34] E Eugene Schultz and Edward Ray. Rootkits: The ultimate malware threat. *Information Security Management Handbook*, 2:175, 2008.

[35] Panda Security. Spam. Available at http://www.pandasecurity.com/homeusers/security-info/cybercrime/spam/ (2013/10/22).

[36] P. Shetty. Rootkits: Both sides of the backdoor. Available at http://www.scf.usc.edu/~shettyp/rootkits.pdf (2013/07/15).

[37] Tyler Shields. Survey of rootkit technologies and their impact on digital forensics, 2008.

[38] Hamza Sirag, Nihant Bondugula, and Rishabh Gupta. Advanced persistent attacks: Bios rootkit-mebromi. 2011.

[39] T. Thomas. What is heuristic antivirus detection? Available at http://www.internet-security-suite-review.toptenreviews.com/premium-security-suites/what-is-heuristic-antivirus-detection-.html (2013/09/30).

[40] Sampo Töyssy and Marko Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2):109–119, 2006.

[41] Truff. Infecting loadable kernel modules. Available at http://www.linux-box.nl/~roeland/doc/phrack61.pdf (2016/02/15).

[42] Unmarshal. Syscall-proxy. Available at http://www.github.com/unmarshal/Syscall-Proxy (2013/11/15).

[43] J Wallen. Five tips for dealing with rootkits. Available at http://www.techrepublic.com/blog/five-apps/five-tips-for-dealing-with-rootkits/ (2013/09/02).

[44] Merrill Warkentin. *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues: Managerial and Technical Issues*. IGI Global, 2006.

[45] Webopedia. What is keylogger?