Back to Static Analysis for Kernel-Level Rootkit Detection

Seyyedeh Atefeh Musavi, and Mehdi Kharrazi
DNS Laboratory, Department of Computer Engineering,
Sharif University of Technology, Tehran, Iran
{*amusavi@ce., kharrazi@*} *sharif.edu*

***Abstract*— Rootkit's main goal is to hide itself and other modules present in the malware. Their stealthy nature has made their detection further difficult, specially in the case of kernel-level rootkits. There have been many dynamic analysis techniques proposed for detecting kernel-level rootkits, while on the other hand, static analysis has not been popular. This is perhaps due to its poor performance in detecting malware in general, which could be attributed to the level of obfuscation employed in binaries which make static analysis difficult if not impossible. In this manuscript we make two important observations, first there is usually little obfuscation used in legitimate kernel-level code, as opposed to the malicious kernel-level code. Second, one of the main approaches to penetrate the Windows operating system is through kernel-level drivers. Therefore by focusing on detecting malicious kernel drivers employed by the rootkit, one could detect the rootkit while avoiding the issues with current detection technique. Given these two observation, we propose a simple static analysis technique with the aim of detecting malicious driver. We first study the current trends in the implementation of kernel-level rookits. Afterwards, we proposed a set of features to quantify the malicious behavior in kernel drivers. These features are then evaluated through a set of experiments on 4420 malicious and legitimate drivers, obtaining an accuracy of 98.15% in distinguishing between these drivers.**

***Index Terms*— Malware, Rootkit, Static analysis, Kernel driver.**

I. Introduction

The concept of malware has evolved over the years, originally written by a single author with a specific functionality, to the current modular malware having several different functionalities. In fact and given the modular design, multiple authors, toolkits, legitimate off-the-shelf modules, or even modules from previous malware may be involved in creation of newer malware. One of the most important modules which increases the threat of malware is the rootkit module. It has been reported by Kapoor et al. [1] that over 10% of current malware contain such model, hence named rootkit. While this may present a small percentage of the total malware population, it is widely accepted that rootkits are far more dangerous than non-rootkit malware. For example, TDSS and ZeroAccess are rootkits which are currently used in the largest active bot networks.

Rootkit may be considered as an independently developed module for hiding the presence of other modules employed in the malware. This has had at least three significant effects on the malware industry: (i) as hiding technologies have been improved, newer malware are less concerned with their size on the victim's hard disk. For example it is reported in [2] that the Zegost rootkit driver takes about a 100 MBytes of storage space on disk; (ii) rootkit modules are included in the Malware-as-a-Service infrastructure. While this underground market is known more for selling automated panels for botnet management [3], it now also offers rootkit modules. In parallel, open source rootkits have became another source for reuse of such modules in combination with several malware. For example the FU rootkit [4] as a process hider in some variants of Rbot[5] and Mytob.r[6] or JiurlPortHide open source driver in ProAgent2.0 spyware [7]; and (iii) The trend to employ rootkit technology in legitimate applications, such as security products [8][9], copy protection technologies [10], recovery tools[11], and third party boot loaders[12]. This has resulted in a negative impact to the detection accuracy of security tools which try to look for evidence of rootkits.

Rootkits could be classified broadly based on their point of operation within the operating system. Processes need to employ APIs or use system services to communicate with the operating system, with which the higher level commands are executed in a lower level language understood by the hardware. Any rootkit which intercepts this communication in user mode is called a user-level rootkit. Example techniques used in user-level rootkits include patching the OS libraries, utilizing OS facilities for hooking processes, or terminating security products through user-level vulnerabilities (e.g. ACL, ActiveX, etc.). As these type of rootkits operate in the user-space, security tools which reside in the kernel-space are able to monitor and detect them. Alternatively, kernel-level rootkits operate in the kernel-space and have much more potent capabilities. There are several types of techniques employed, including file masquerading, redirecting the execution path of the kernel code by hooks and patches, DKOM (i.e. Direct Kernel Object Manipulation), installing a malicious filter driver, changing the sequence of boot procedure (including MBR code or partition table, VBR IPL code, BIOS Parameter Block modification or even BIOS code)[13], use of HFS (Hidden File System)[14], and finally virtualization and other firmware modification methods[15][16]. The reader is referred to [17] for a good review of rootkits.

A great number of solutions have been proposed for ordinary malware detection in the literature, which mostly operate

in the user-level and are not effective for detecting kernel-level rootkits. This is mainly due to the fact that these detection techniques are able to view events at their level of operation (i.e. user-level) and trust the operating system for gathering required information about the system; Where as kernel-level rootkits are able to hide their presence to the operating system and in turn the user-level processes.

Nevertheless, there are a number of proposed solutions for detecting kernel-level rootkits. One of the more prevalent approaches is to detect the rootkit by executing it in a hardware/software virtualized environment. But such approaches are not widely deployable on the end user's system due to resource requirements, performance issues, and transparency challenges. Alternatively there are a number of techniques which operate within the operating system, and execute the detection procedure through dynamic analysis. Such approaches are not without risks and shortcomings. First the suspicious binary has to be executed on the system in order to be analyzed. Second, all possible binary routines may not be executed as they would require specific conditions to be available (e.g. not getting executed in a sandbox), hence the binary may not be analyzed completely. In other words, unless the binary is executed in the proper environment, it will not show any malicious behavior. An alternate method of detection which operates within the operating system is through static analysis, where the binary is analyzed as a whole and malicious behavior/patterns are detected regardless of whether those sections are executed are not. Nevertheless, this approach has not been popular nor successful with malware in general, mainly due to the fact that obfuscation techniques are widely available and employed, making such analysis quite difficult.

There are two important observations which should be considered when discussing kernel-level rootkits as compared to user-level rootkits and malware in general. First, the key role kernel drivers play in rootkits, with which the rootkit penetrates into the Windows kernel space. Second and perhaps more importantly, while powerful obfuscators (i.e. packers or encryptors) are abundant and used regularly for either malicious or legitimate purposes in user-level executables, they are not as widely available nor used in kernel-level code, at least for legitimate purposes. This could be explained by the fact that there are a smaller set of facilities with which one could hide the intent in kernel code, specifically in low-level assembly code, and the difficulty that obfuscation brings when maintaining and debugging code in the kernel-level. In other words, even though malware developers may employ obfuscation, there is much less motivation for legitimate driver developers to use obfuscation. Hence, the presence of obfuscation could indicate potentially malicious behavior.

Considering the above noted observations, while taking into account the issues and limitations in dynamic analysis techniques, we propose a static analysis technique which could be used to detect rootkit drivers in the Windows operating system on an end-user machine and can complement previously proposed dynamic analysis techniques. Another more important usage of such static analysis technique would be to help in extracting rootkit driver samples from a large corpus. Usually and in order to decide on the nature of collected samples, the samples are executed in a controlled environment (i.e. sandbox) and with the help of dynamic technique the samples is analyzed. This process is quite time consuming as each sample is to be executed properly. Furthermore and at times the malicious samples intentionally increase the time of execution, for example by adding sleep timers. Therefore the detection technique would not detect any malicious activity in finite time and could classify the sample as legitimate. But static analysis techniques, such as the one noted in this work, would only have to parse the code and calculate a set of features, with out any need to execute any samples. This process would be more efficient and resistant to environment aware malware.

Therefore, we propose a set of features to distinguish between malicious and legitimate drivers based on a study of modern kernel-level rootkit behaviors. Furthermore, we evaluate the proposed features by gathering 2200 kernel-level rootkit drivers and 2220 legitimate drivers and obtain and accuracy rate of 98.15% in distinguishing between the legitimate and rootkit drivers. In fact, and to the best of our knowledge, other that Kruegel et al. [18] which look for improper memory access as a sign of rootkit activity, there has been no prior work which employs static analysis to extract pre-defined behavioral features covering different kinds of kernel-level rootkit drivers. It should be noted that we focus on detecting kernel-level rootkits and consider user-level rootkits as out of scope. More specifically the main contributions of this work include a study of behavioral trends in current day Windows kernel-level rootkits and then proposing a set of features to differentiate between their malicious behavior and that of the legitimate drivers.

In the remainder of this paper, we first review the related works in Section II. The proposed static analysis based detection technique is presented in Section III, and in Section IV we discuss the implementation and evaluation of the proposed detection technique. We discuss a number of related issues in Section V and conclude this manuscript in Section VI.

II. Related works

Many rootkit detection techniques have been proposed in the literature since 1999, when the first known rootkit (i.e. NTRootkit for the Windows operating system) was created. There has been much published work on the hardware/software virtualization based detection techniques [19][20][21][22][23], but there are limitation in employing such techniques in practice and on an end-user system. In addition to performance penalty, detection of the emulation environment by the malware, as well as special hardware requirements in the case of hypervisors, a number of studies [24][25][16][22] show that such environments have their own vulnerabilities and challenges. Alternatively, a number of proposed detection techniques operate from within the operating system and require no artificial environment. Kernel Integrity check [26][27] or crossview [28] are two common dynamic detection approaches. Most of these proposed schemes focus on a limited category of rootkits and are unable to detect a wide range of

rootkits. In the rest of this section, we will focus on related works on static analysis as well as a number of techniques which have focused on malicious rootkit driver detection as they relate more closely to this work.

There have been different methodologies proposed for static detection of malware in general. These techniques could be categorized, based on the approach used to define the distinguishing features, as either blind or behavior driven. There are a large number of blind techniques proposed in the literature. For example, Byte-level signatures [29][30] are widely used in anti-virus products. In [31] the authors argue that byte level signatures would be ineffective in the presence of polymorphism and mutations and instead focus on features based on opcodes. An alternate approach is to use the type and number of system calls found in the code as features to distinguish legitimate and malicious samples. Schmidt et. al [32] gather function calls observed in the code and selects a small set as distinguishing features based on statistical analysis. Sami at al. [33] use PE header imported functions to find the frequent API call sets which are then used as features. Such approaches result in a large number of features which should then be reduced for efficient classification.

As an extreme example Dahl et al. [34] extract 50 million features from 2.6 million malware samples before reducing the number of features. More specifically, authors in [33], [34], [35], and [36] have employed the fisher score, random projection, information gain, and feature-hashing respectively, in order to decrease the large number of features obtained and select the more important features. These approaches do provide acceptable accuracy results by only considering system calls, while ignoring the arguments and parameters used in the call, nevertheless use of obfuscation would certainly affect their accuracy.

On the other hand, there have been a number of studies in which the features are proposed based on some initial knowledge of malware's behaviors. Prophiler [37] is one such work in which 77 features are proposed for detecting malicious behaviors in webpages. As another example Zhao et al. [38] extract a FCG (i.e. function call graph) from a file and propose 32 distinguishing features related to the structure of the graph and defined function. In such approaches, the features are defined and selected based on a perceived malware behavior as opposed to a statistical feature selection process.

Another important approach proposed in works such as [39] and [18] is semantic-aware detection. For example Kruegel et al. [18] concentrate on the detection of kernel-level rootkit drivers by modeling improper kernel memory accesses. More specifically, the sample code is executed symbolically and when a sequence of instructions match that of the model, malicious activity is identified. In fact and to the best of our knowledge, this is the only static analysis technique which focuses on kernel-level rootkit drivers.

We should also note two other important detection techniques which focus on kernel-level rootkit drivers, although they are not purely based on static analysis. dAnubis [40] is an extension to Anubis [41] which analyzes kernel driver behavior by Virtual Machine Introspection (VMI) and provides a complete overview of how it communicates with other elements in the system, and its interaction with the system memory. However, the main goal for dAnubis is to provide a human readable report on the driver and it does not provide a detection service. Limbo [22] is another proposal which focuses on detecting kernel-level drivers. It extracts a set of behavioral features dynamically in an extended PAM32[2] emulator, where they are mostly related to popular hooks and DKOM techniques as well as general properties they found in rootkits. Furthermore a few general static features from the driver's PE header are also considered. They evaluate the proposed technique with a total of 754 kernel-level drivers and obtain and overall accuracy of 96.2%. In the next section we will introduce the proposed static detection technique.

III. Static Detection of Rootkits

As noted briefly in Section I, static analysis is thought to be insufficient to perform typical malware detection in user space. Nevertheless, there are a number of issues which suggest that such approach would be more effective when looking for malicious kernel-level drivers and in turn kernel-level rootkits. One could observe that two issues have resulted in a shift from the static analysis towards the costly dynamic analysis in the malware detection techniques. First, and as noted by Moser et al. [42], obfuscation techniques are being widely deployed, which makes low cost static analysis inefficient if not impossible. Second, malware detection techniques trust the operating system for run-time analysis by default. Given such trust, dynamic analysis is easier as compared to the difficulties with reverse-engineering of malware code for static analysis. Neither of these two issues are valid when confronting rootkits.

Control-flow obfuscation, run-time load of OS modules, and command, callee, and string coding/encryption are the most common approaches for code obfuscation in an executable. Control Flow obfuscation is harder in kernel space because of the modular nature of a driver structure (a DriverEntry and some IRP [3] handler routines) with determined signatures.

As for run-time load of OS modules, while user-level malware prefer to dynamically load API functions instead of static linking to obfuscate their code, there are small number of kernel functions with less usage/power (e.g. callingMmGetSystemRoutineAddress, MmLoadSystemImage, etc.) which allow loading kernel modules in run time. This makes obfuscation in kernel code more difficult. Another important point, as noted in [43][44], is that obfuscated code executes slower than normal code. Assume a filter driver which spends considerable time for de-obfuscating its code when it's called to filter the results of a disk or network query. The lazy kernel response to each request would not be transparent. This is while small performance penalty and automated procedure of deploying such techniques in user-level binaries by packers and encryptors, makes obfuscation common in both malicious and legitimate binaries at that level. Some of these techniques can be implemented in kernel space, but with more constraints and limitations on facilities. Most of the existing constraints are related to the stability and performance limits in the kernel

[2] An X86 ISA emulator employed mostly by Symantec.

[3] I/O request packet

memory space which is shared by many other kernel modules and the operating system itself.

Furthermore, the need for maintenance, debugging, and crash dump analysis of drivers required for legitimate development tasks encourages developers not to employ obfuscation techniques. Exceptions do exist, for example DRM enabled[45] and copy protected drivers (i.e. for gaming, audio, and video products) do use obfuscation at the cost of less stability and slower execution time as discussed in [46]. All above issues results in much limited possible volume of obfuscation in kernel modules. In practice legitimate driver developers do not generally employ such techniques, although as noted there are exceptions. This is while malicious driver developers, as a best practice in their respective domain, try to employ these techniques to make the analysis more challenging. Thus the presence of obfuscation would potentially point to malicious intent in the code.

The second issue noted is the assumption that the OS could be trusted. In dynamic detection inside the OS, the detector has the same power and privileges as the rootkit. Thus there is no guarantee that it would be able to monitor the system completely. Hidden processes, firewall bypass, and Kernel Hook Bypassing Engine attacks[47] are examples of such threats. Migrating this run-time analysis to outside of the OS through hardware extensions or hardware/software virtualization bring about other resource requirements and limitation which make such approach difficult and at times costly to apply on an end user system.

Considering the noted issues, we propose a simple light weight static detection technique. The detection process is invoked when it is called to scan all kernel-drivers on disk or when the memory monitor indicated that a driver is dropped/installed by a user-level application. The kernel-driver is then dis-assembled and a set of features extracted. Finally the analyzer agent, which is a binary classifier, classifies the driver as either malicious or legitimate. In what follows we first discuss the trends we have observed in kernel-level rootkits in Section III-A. The proposed static features are then introduced in Section III-B.

A. Trends

Based on the malicious behavior of rootkits expounded in [48][49][50][51][52], large number of studied reports obtained from different malware analysis labs (i.e. Kaspersky, McAfee, Eset, Symantec, etc.), and static analysis done on a number of sample drivers, we observe a range of trends in the functionalities provided by rootkit drivers. These functionalities include injection, hooking, DKOM, kernel memory over-writing, system modifying (modify system by installing filter-drivers/devices, providing new hidden file system, etc.), loading (load files or stored binary data into memory), and networking. Where a malicious driver may have one or more of the above noted functionalities implemented. These trends, when considered against documented facilities for legitimate drivers [53][54][55], could be used to distinguish between rootkit and legitimate drivers. In what follows, we discuss some of the more important trends observed:

1) Injection: Injection or more generally patching is one of the most popular behaviors in rootkits. A malicious kernel driver may patch user-level executables, operating system kernel modules, or other drivers. One of the most common methods to inject into user-level processes is to attach to a target process and inject a desired code via allocation of Virtual Memory in context of a newly created or existing thread. Another technique includes creating and mapping of a new section into a process memory space. A more stealthier approach is to use APC [4], in order to run a thread or raw code in context of a target process. Usage of these techniques from kernel space make patching stealthier, although these functions often have their user-level alternatives. There are also kernel specific facilities for injection. Kernel thread injection, as explained in [56], is an example with which the malicious functionality is executed through kernel space. For patching kernel codes there is no facilities from user-level programs in current versions of windows(i.e. XP SP3 and later). Thus Shared kernel memory available for kernel drivers makes it possible to overwrite existing codes via accurate kernel memory allocations. It is important to note that because injection is one of the prevalent behavior in large number of rootkits with rare usage in legitimate drivers, it can be considered as a good delimiter to distinguish malicious from legitimate drivers.

2) File Activity: Programmers are discouraged [57][58] from using file activities in kernel drivers in some kernels to avoid programming errors and subsequent system crashes, as well as difficulties in applying policies in kernel-level. While kernels like Linux are more restrictive to such activities, Windows does include some function calls for accessing files for situations like when a driver is going to update hardware microcode by downloading new code [53]. However the procedure should be handled carefully, for example and as pointed in [59], if a driver is loaded before determination of drive letters, the DosDevices namespace which is used to access file objects may not exist. Nevertheless, there are situations in which a rootkit requires to obtain a handle to modify a file including logs and/or spying data, or alter file access times. Thus we believe that such behavior should be observed more in rootkit drivers rather than legitimate drivers.

3) Malicious vs. legitimate filters: Attaching to a target device is a common behavior in rootkit filter drivers and IRP [5] hookers. However such technique is also employed in legitimate drivers which operate in the layered Windows architecture, like multimedia and modem device drivers. The point is that malicious filter driver needs to log the events in order to know when the lower level process has completed. However there is a technical limitation as the filter is run in the DPC [6] level, and file access via the ZWxxx routines requires the filter to be executed at the PASSIVE LEVEL[53]. Thus and in order to have the proper file access, another system thread is executed in PASSIVE level to run ZwWriteFile. Observing code which is used to notify this system thread could be taken as a sign for identification of such filters.

[4]Asynchronous procedure call
[5]I/O Request Packets
[6]Deferred Procedure Call

TABLE I
THE PROPOSED FEATURES, GROUPED INTO FIVE CATEGORIES

Feature category	Features
General behavior	Allocations, handled MJ IRP count ,dispatch routines, network activity, file activity, registry activity, file system activity, hardware activity, synchronization (i.e. wait/signal, Set timer, alert, set event, etc.) , exclusive access (i.e. mutex, fast and guarded mutex, locks, semaphors, etc.) , RaiseIRQL, DPC level activity (i.e. whether a driver works at dispatch level or not) , deploying system thread, attach devices, device activity, memory overwrite, port level activity
Communications	user communication (i.e. SymbolicLink, DeviceInterface, operation on VirtualMemory of user-level processes), kernel communication (i.e. kernel APC, load image, load filter, set information of jobs, processes, etc.)
Rootkit-like behaviors	injection, general hook (heuristic to follow unknown hooks as much as possible) ,SSDThook, IRPhook, DKOM (heuristic to estimate usage of functions/constants which are used in XP rootkits) , filter driver, bootkit-like (activities related to boot time, BIOS, etc.) , write protection (CR, MDL or VirtualProtect)
Overall static feature	device counts, DriverEntry size, number of system calls , DriverEntry subroutines, constants, dis-assembly size, strings, string activity ratio, allocation to dis-allocation ratio
Suspicious behaviors	track process, query about a specific file, notification installing (i.e. callbacks or notifies), anti-analysis, environment aware, undocumented devices, non-English characters, dynamic load, system reconnaissance (i.e. trying to identify hw/sw specification) , own network stack, security descriptor modification, misc-suspected, minimum per character entropy, average score of all attributes

4) Bypass memory write protection: There are two different ways a rootkit may choose to bypass write protection for kernel code pages it wants to hook or inject: CR0 register modification and MDL[7] modification, which are discussed in [49]. While modifying the write protection bit of CR0 is a good indicator of malicious activity, creating a MDL is a common behavior where its argument, which describe the desired region of memory, can determine whether it is malicious or not. Using VirtualProtect function is another technique a rootkit employs to modify the access protection of desired pages of memory in order to change code segments. In such scenario it is required to ensure that the instruction cache does get updated with new instructions by calling FlushInstructionCache. This behavior, although not advised[60], can be used in some legitimate scenarios such as self-modifying codes like packed/compressed executables and other applications such as real-time graphic drivers.

5) Track Process: Tracking a process on the memory has many applications for a rootkit. DKOM drivers are mainly module hiders which track the target executable in the memory, use the result specification as an index to find the target in EPROCESS list, and then change the pointers in order to hide the module, details of such operation is discussed on [48]. Furthermore, injectors also should find their decoy process before injection. Any protector driver (hookers or filters) also requires to track their subject module to get its specification, so that it could filter related commands for the module. This means that unlike legitimate drivers, which often communicate with a module they have looked for on memory (e.g. by APC), malicious drivers continue the tracking procedure either by injections or by in-direct communication with the module (i.e. by DKOM and hooking).

6) Own network stack implementation: As noted by Vieler [51], C&C communication is one the most revealing points for malware to be detected. This is because deploying Winsock Layered Service Provider (LSP) DLL of Windows in user-space is easy to use but also easily visible. Hence, rootkits try to minimize their visibility by employing the TDI and NDIS interfaces in the kernel space. Since TDI interface operates at a lower level than that of the security monitors with user-level sockets, many rootkits prefer to use it. NDIS driver is the lower interface which allows the process, either malicious or legitimate, access to raw packets. By dealing with raw packets at this level, one can bypass many existing security products such as firewalls. As reported by Kasslin [61], Srizibi was the first malware to bypass many firewalls by the use of this technique. Thus one of the most suspicious behaviors for a driver is to use its own network stack via NDIS library instead of implemented OS stack at this layer.

In the rest of this section we will discuss the proposed feature set which would help in distinguishing between malicious and legitimate drivers.

B. Driver features

Based on the behavioral trends observed, some of the more important of which were reviewed above, we propose 50 features to distinguish between legitimate and malicious rootkit drivers. These features could be classified in five broad categories as noted in Table I and described below:

1) General behavior: The set includes different possible activities on files, registries, and network as well as some limited measures for file system activities. It also enumerate any exclusive accesses (e.g. Mutexes, SpinLocks, CriticalRegion, etc.), synchronization processes (e.g. timers, waits, and events), and increase in DPC or more generally IRQL levels. This category reveals the overall functionalities of the driver.
2) Communications: Two separate sub-categories were defined, one for kernel and another for user-level communications. For the kernel-level communications we have considered image loads, kernel APCs, and other kernel functions which allows the driver to modify any kind of system objects such as ZwSetSystemInformation. User-level communication is considered by looking for signs of SymbolicLink or DeviceInterface being deployed.
3) Rootkit like functionalities: Based on different types of rootkits analyzed, the following categories have been defined to show the related activity of the driver: injection, general hooking behavior, as well as special hooks like

[7]Memory Descriptor List

SSDT and IRP hooks, DKOM, bootkit-like behavior, filter drivers, and bypassing write protection mechanisms in memory.

4) Overall static features: These features are related to the data we can extract from a driver binary regardless of its context or functionality. The main goal of this class of features is to provide a measure of obfuscation employed in the driver. Number of kernel-function calls, size of DriverEntry, device counts are some examples of these features. Apart from these, number of strings and the ratio of string-related call activities to the overall size of driver are other examples used to get a view about probable string obfuscation employed in the driver.
5) Suspicious behaviors: While the first three classes were necessary to find kernel-level rootkit drivers, they are not sufficient. A driver may use similar technologies for a legitimate task. The idea here is to involve other suspicious behaviors as some evidences of being malicious. These features have been selected in a diversified manner to cover multiple suspicious activities. Tracking a process in memory, query about a specific file, anti-analysis behavior, obfuscation (i.e. small or nested DriverEntry routine, use of special characters like *, using familiar looking device names, etc.) are examples of this category of features. While callbacks and notifiers are facilities for any kernel driver, they have vast usages for rootkits. Thus we have considered multiple kind of calls from popular "ExCreateCallback" to less known "ZwAccessCheckByTypeAndAuditAlarm". In addition since rootkits operate deep in the system, they need to query about some system specification in hardware or software levels. Hence we define a system_reconnaissance feature to cover such software checks from OS version to user interface language or environment variable checks. multi_processor is another feature engaged with queries about max or active number of processors or other related functions.

Our feature set has a multi-aspect approach to the problem of distinguishing between malicious and legitimate drivers. For example assume a SSDT hooker driver, which uses dynamic load techniques to hide system calls it requires to employ. With this example, the SSDThook feature will provide little information, however the feature which looks for dynamic loads will provide valuable distinguishing information. As another example, we consider obfuscation in some of the proposed features (i.e. 10 features are obfuscation related). Therefore, if a driver uses obfuscation, it will obtain bigger scores in obfuscation related features and lower scores in other features. We consider this difference in the scores as a distinguishing factor. In other words the proposed feature set tries to look for any abnormal combination of events/behaviors employed in the driver which could help in detecting known behaviors in an unknown rootkit. In the next section we will evaluate the proposed features.

IV. Implmentation and Evaluation

In this section we first introduce the dataset gathered for evaluating the proposed static analysis detection technique in Section IV-A. We then discuss how the proposed features are extracted in Section IV-B. Given the collected dataset, functionality trends noted earlier are analyzed in Section IV-C. Lastly, we evaluate the proposed features in Section IV-D.

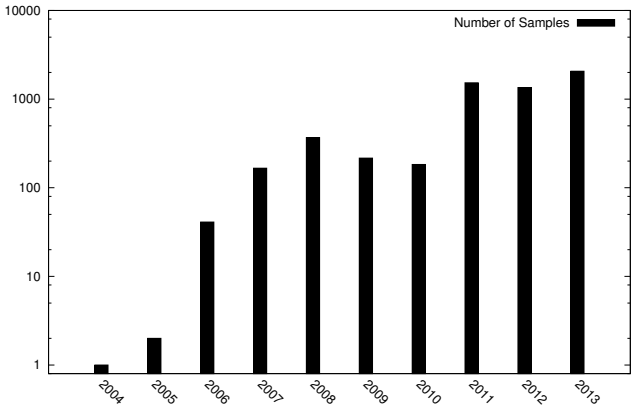

Fig. 1. Distribution of samples collected based on their submission year to the malware repository. Note that the Y-axis is in log scale. Furthermore, there was only one sample in the data set which was submitted to the malware repository in 2004.

A. DataSet

In order to evaluate the proposed detection technique and its ability to distinguish between legitimate and malicious drivers, we gathered two datasets, one with legitimate drivers and one with kernel drivers used by malware. As for the legitimate driver dataset, we collected 1600 distinct drivers from three driver collection products (i.e. Universal XP Driver pack, Huge Drivers Collection Pack for Windows XP, and Driver Pack for windows 7 and Vista) plus 600 XP SP3 drivers on dllcache folder of an XP system. These drivers include many low level filters, such as network, sound, video card, wireless, webcam, blue-tooth, and other application drivers from different vendors.

Unlike extracting driver modules from normal applications, extracting such a large number of drivers by running rootkit samples in an isolated environment is a challenging process. This is because of a number of issues unique to the kernel-level rootkits which includes kernel-level implementation of Anti-VM techniques, usage of DKOM techniques to hide modules, forcing a reboot on the infected machine, denying access to modules by kernel-level hooks and protection, and storage of driver modules in hidden file systems. Thus we obtained module samples with "sys" extension from the VirusShare [62] malware repository. It should be noted that under the McAfee naming scheme, that extension would correspond to malicious drivers used in current day rootkits. Figure 1 provides an overview of the distribution of samples collected based on their initial submission time to the malware repository.

To check the diversity of the malicious set we have employed the Ssdeep clustering tool [63] on 5000 downloaded malicious drivers. Ssdeep is a popular implementation of CTPH [8], which can identify files similar to each other and

[8]context triggered piece-wise hashing

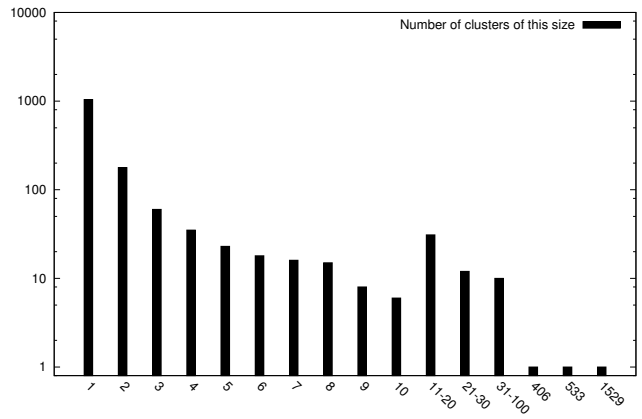

Fig. 2. Malicious drivers are clustered based on their similarity. The distribution of cluster sizes is illustrated. In one extreme, there were a total of 1042 clusters with each with a population of one driver, in other extreme there is a single cluster with a population of 1529 drivers. Note that the Y-axis is in log scale.

is used for locating similar pieces of malicious code. As expected, there are some drivers which are being used by multiple rootkits. These similar drivers are grouped into clusters, where Figure 2 illustrates the cluster size distribution. As we had 2200 legitimate drivers, we randomly selected 2200 malicious drivers from 1459 distinct clusters, where the number of samples used from each cluster was relative to the cluster size. This method of selection insures that samples from the more popular rootkit drivers would also be more present in the dataset. It should be noted that we also used the CTPH utility on the collected legitimate dataset, where there were 1978 clusters. 1934 of these clusters included only one sample. This ensures that the legitimate data set is not populated by specific types of drivers.

Additionally, we also added some drivers from software products which are likely to cause false positives in the detection routine, such as those discussed in section I. This was done by collecting 20 drivers used in such applications, including TrueCrypt, Safedisk, Kaspersky, Returnil AV, Comodo recovery, PC-back, Sentiel, Daemon Tools, Dwall, Malwaredefender, and VMware. Hence, we have a total of 2220 legitimate and 2200 malicious drivers which we used in the evaluation.

B. Feature extraction

A number of distinguishing features were proposed in the previous section. These features, some of which are abstract, were calculated by looking at smaller elements in the driver's dis-assembled code. Simply put, the number of instances an element related to a certain feature is observed in a dis-assembled driver, would make up the value for that feature. The elements considered are:

- Kernel function calls: There are a large number of kernel functions (i.e. Zw, Ke, Ks, Cm, Ex, Hal, Io, Mm, Ob, Po, Ps, Rtl, Se, Ndis, and CRT) in Windows including either documented or undocumented ones. These functions and libraries can disclose the general intent of a module. These functions are clustered, where each cluster is tied to a functionality.
- Constants: Constant values in dis-assembled code provide important information about the code being analyzed. For example, special offsets of kernel memory are often used to overwrite on (e.g, the offset for MBR).
- Assembly commands : There are some special assembly instructions which are used by rootkits for known intents. Examples include instructions to modify CR register for bypassing write protection of kernel or looking for the INT3 instruction, with which the presence of a debugger is detected and therefore the executable presents a non-malicious behavior.
- Type and value of variables: One of the sources for finding special functions between all subroutines is to check their arguments type. As an example one way to determine the dispatch routines in a driver (beside other heuristics) is to look for two arguments to be DEVICE_OBJECT and PIRP.
- Calculated measures: There are measures with which the appearance of the code is quantified. For example, minimum per character entropy, number of nested routines in DriverEntry, or size of the dis-assembly. In fact, there are 4 features[9] which are only dependent on these measures. In other words, they could be calculated properly regardless as to what percentage of the driver is successfully dis-assembled.

TABLE II
NUMBER AND TYPE OF ELEMENTS USED IN CALCULATING THE FEATURES FOR EACH CATEGORY

Feature category	Kernel functions	Constants	Assembly commands	Type and value of variables	Calculated measures
General	13	2	1	1	1
Communications	2	-	-	-	-
Rootkit functionalities	7	1	2	3	1
Overall static feature	1	2	1	3	7
Suspicious behaviors	10	3	-	3	2

Table II provides an overview of how the above element categories are used in calculating the five feature categories noted in Section III. It should be noted that a feature could be calculated by considering elements from more that one element category. In order to clarify how the features are calculated, we will describe the calculation of five features in more detail, where their pseudo-code is available in Figure 3. The "DKOM feature" is calculated based on the first and second category of elements noted above. This is done by enumerating system calls which are used by rootkits to obtain required kernel objects as well as the number of accesses to the offsets of such objects (i.e. Flink,Blink, or EPROCESS). The number of instances these system-calls and offsets are observed would be counted and then the value is normalized

[9]Dis-assembly size, Minimum entropy, Number of detected system calls, and Average score of all features (as lesser parts of a driver are dis-assembled, the average would be lower.)

Fig. 3. pseudo-code for calculating the DKOM, DPC level activity, Dispatch routine, Device activity, and undocumented device features

$DKOM\_cluster \leftarrow$(PsActiveProcessHead, PsLoadedModuleList, PsLoadedModuleResource, ExCreateHandleTable, ExDupHandleTable, ExSweepHandleTable, ExDestroyHandleTable, ExChangeHandle, ExSnapShotHandleTables, ExfInterlockedInsertTailList, ExfInterlockedRemoveHeadList, IoGetDeviceObjectPointer)
$DPC\_cluster \leftarrow$(KeRaiseIrqlToDpcLevel, KeInitializeDpc, KeInsertQueueDpc, KeRemoveQueueDpc, KeSetTimer, KeSetTargetProcessorDpc, KefAcquireSpinLockAtDpcLevel)
if $CallExistInDKOM\_cluster$ **then**
  $DKOM\_f \leftarrow DKOM\_f + 1$
end if
if $OffsetExistIn$(Flink,Blink, EPROCESS) **then**
  $DKOM\_f \leftarrow DKOM\_f + 1$
end if
if $CallExistInDPC\_cluster$ **then**
  $DPC\_f \leftarrow DPC\_f + 1$
end if
if $SubRoutine$ AND $FirstArgType == DEVICEOBJ$ AND $SecondArgType == PIRP$ **then**
  $Dispatch\_f \leftarrow Dispatch\_f + 1$
end if
if $variable_Type ==$DEVICE_OBJ **then**
  $Device\_activity\_f \leftarrow Device\_activity\_f + 1$
  if $Device_TypeExistIn$58_system_defined_devices **then**
    $undocumented\_device \leftarrow undocumented\_device + 1$
  end if
end if
$DKOM\_f \leftarrow normalized(DKOM\_f)$
$DPC\_f \leftarrow normalized(DPC\_f)$
$Dispatch\_f \leftarrow normalized(Dispatch\_f)$

to be used a the feature value. The "DPC level activity" is also calculated in a similar manner. The "Dispatch routines" feature is calculated by counting the number of subroutines with two argument types of: DEVICE_OBJECT and PIRP. As another example, the "Device activity" feature is obtained by counting the variables with the DEVICE_OBJ type. Furthermore the number of times a device type does not exist in the Microsoft pre-defined devices, constitutes the "Undocumented devices" feature.

Based on the above noted elements, a feature extractor is implemented in the Perl scripting language which take as input a dis-assembled driver. Dis-assembly is done the by the powerful IDA dis-assembler [64]. The current feature extractor implementation is available at [65].

C. Trend verification

A number of trends were observed when analyzing rootkit drivers as noted in Section III-A. We employed the collected dataset, in order to investigate the generality of these trends. The obtained results are noted below:

1) Injection: As expected, injection is more widely used in malicious driver modules. More specifically, more than 27% of malicious modules had in one way or another signs of injection being used, while there were only 1.4% of legitimate drivers with similar functionality. Although small, we did not expect to see legitimate drivers in which injection was being employed. After a closer examination of these drivers, we have found some legitimate behaviors which are similar to injection. For example, some of the drivers attach to the Crss.exe system process in order to copy BIOS from its memory, this is instead

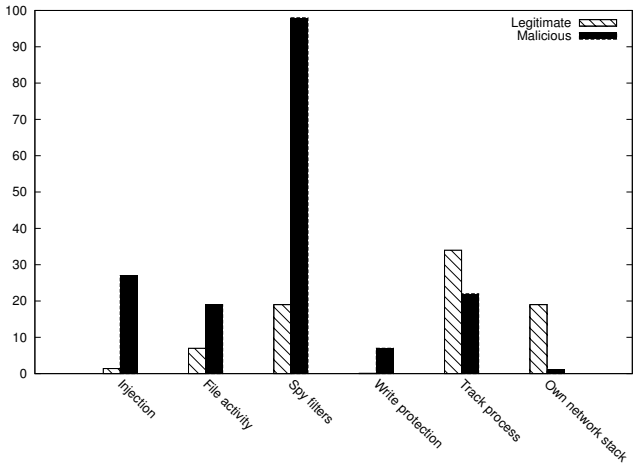

Fig. 4. An illustration of the five noted trends as obtained from the collected malicious and legitimate driver samples.

of properly accessing the BIOS through its real offset value. Another example was calling the ZwMapViewOfSection function for mapping memory on to a process from the kernel, done by legitimate mini port drivers [66] which is similar to the injection behavior.

2) File Activity: The result shows that while 19% of malicious drivers have had file modification activity, this was done by only 7% of legitimate drivers. This was broadly because audio device drivers create a file in order to communicate with the client using KS[10] filters[67].

3) Malicious filters: 98% of malicious drivers showed file modification activity, while 19% of legitimate drivers showed similar activity. Again, mostly legitimate audio drivers were showing such activity. Furthermore and as expected, a larger correlation exists between the feature for file modification and two other features, installation of notification facilities and deploying system thread, in malicious filter drivers than legitimate ones.

4) Write protection: In practice 7% of the rootkits bypass memory protection using MDL or CR0 modification in order to hook SSDT. There were also a total of three legitimate drivers, belonging to Anti-Virus and firewall products with similar behavior.

5) Track process: As expected, tracking processes is a normal behavior in legitimate drivers as well as malicious ones, such that 22% of legitimate drivers and more than 34% of malicious modules show signs of this. Thus further communications (i.e. legitimate communication via APC or illegal injection), could show whether this behavior is normal or not. In fact, and on average, legitimate drivers show a higher rate of user-level communication rather than malicious drivers, while having a lower probability of injection or hooking behavior compared to the malicious drivers.

6) Own network stack: About 1.12% of our malicious samples use the NDIS library functions in order to bypass security products and security monitors running on the system. This is while 19% of legitimate network adapter drivers have

[10]Kernel streaming

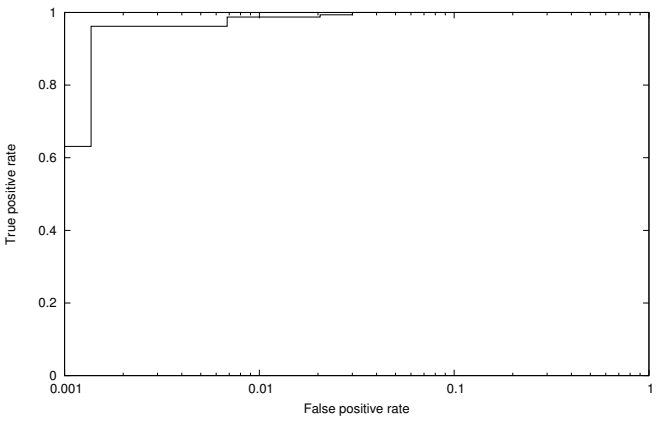

Fig. 5. Calculated ROC curve for the trained C5 classifier using our test dataset. The area under the ROC curve is 0.99.

TABLE III
TOP 26 FEATURES RANKED BY PEARSON FACTOR

Rank	Feature Name	Rank	Feature Name
1	dynamic load	14	constants
2	string activity ratio	15	environment aware
3	user communication	16	strings
4	number of system calls	17	write protection
5	dis-assembly size	18	Track process
6	filter driver	19	non-English characters
7	RaiseIRQL	20	awakening
8	misc-suspected	21	allocation to dis-allocation ratio
9	DriverEntry subroutines	22	entropy
10	anti-analysis	23	device activity
11	SSDThook	24	IRPhook
12	injection	25	undocumented device
13	allocations	26	query about a specific file

much larger score in this feature with larger dis-assembly size. This could be largely attributed to the fact that malicious drivers do not need to completely handle all complexities and implement details of the network protocols. In fact, this difference in the implemented code size could be used as a differentiating factor.

D. Experimental evaluation

In order to investigate the effectiveness of the proposed detection scheme, we trained and tested a classifier. To that end we employed the C5 Tree classification algorithm and used cross-validation with k=10. Furthermore, the classifier identified the more important features using the pearson factor [68], with which 26 features are selected. The results were encouraging. We were able to classify the malicious and legitimate drivers in the test dataset, with an accuracy rate of 98.15%, false negative of 0.6%, and a false positive of 3%. The ROC plot obtained for the classifier is shown in Figure 5, where the calculated AUC is 0.99. Table III presents the 26 features employed, ranked by their importance factor. We have also experimented with a smaller number of features for training and testing the classifier. We obtained an accuracy of 95.64% and 97.9%, when employing the top 10 and 15 features respectively.

Even though we obtained acceptable accuracy results, we wanted to inquire as to why some of the test data was miss-classified. Overall 80 drivers where miss-classified, including 14 false negatives and 66 false positives. Upon a closer investigation, it was found that most of the false-negatives are process-hider drivers like HideProc variants which show the need for more accurate behavior modeling. Failure in dis-assembly [11] was another cause of such errors. Furthermore, False positives were legitimate drivers which were dis-assembled partially in a way that IDA could not find system call names but many constants (which are raw bytes).

As noted in Table III, features such as string activity ratio, anti-analysis, number of system calls, dis-assembly size, constants, and other similar feature are ranked high. These features are mostly used for measuring the level of obfuscation in the drivers. Given these results, we conjecture that the malicious status of a driver neither depends on its general behavior nor the rootkit like operations (e.g. hooking, etc). What is important is the level of suspicious activity present, such as hiding intent (e.g. obfuscation). This could be explained by observing that currently rootkit have benign counterparts with similar behavior and legitimate applications. For example, bootkits vs. bootloaders, spyware filters vs. legitimate filters, malicious hookers vs. security product hookers, and DKOM rootkit hiders vs. game protection products, etc. In what follows we will discuss the obtained results and compare and contrast the proposed technique to the related works in this area.

V. DISCUSSION

Even though the proposed technique is able to detect malicious rootkit drivers accurately, such approach will not be effective against all possible types of rootkits. Zero day exploits on the kernel code or un-patched vulnerabilities could be employed by the rootkit to penetrate into the kernel space without deploying any driver. Nevertheless, given the difficulty in finding such exploits, most rootkits found in the wild prefer to deploy using malicious drivers. There are also few modern rootkits like Festi [69] which use memory resident kernel drivers downloaded from C&C on each reboot. Another similar scenario are rootkits which store the drivers in hidden file systems and load them into memory on each boot. We should note that any kind of penetration into kernel space, other than employing kernel vulnerabilities, requires at least an initial kernel driver to be loaded[12] and then through which other drivers will be loaded. Hence, we can detect the loader kernel driver at the beginning of the process.

Undoubtedly, like other inside OS solutions, deploying the proposed static analysis technique on an actual system requires a number of assumptions. More specifically we assume that the detector, which is a user-level agent, analyzes any new

[11] A total of 9 legitimate and 12 malicious drivers failed to be dis-assembled even in part.

[12] An exception to this would be Windows XP in which a user-level process can modify kernel memory region directly. This issues was resolved in the Service Pack 3 update.

TABLE IV
A SUBJECTIVE COMPARISON BETWEEN THE PROPOSED TECHNIQUE AND TWO PREVIOUS PROPOSALS WHICH FOCUS ON KERNEL DRIVERS AS PART OF THE ROOTKITS DETECTION/ANALYSIS PROCESS.

Capability	Proposed method	Kruegel[18]	dAnubis [40]	Limbo [22]
Does not require run time analysis	Yes	Yes	No	No
Covers multiple rootkit behaviors	Yes	No	No	No
Distinguishes between rootkits and their benign counterparts	Yes	No	-	No
Final decision as to malicious or not	Yes	Yes	No	Yes
Operating system supported	Windows	Linux	Windows	Windows

driver being loaded on the system and itself is protected by Kernel-level self-protection solutions such as [70], which can be employed to insure proper execution of the analyzer. Additionally, kernel integrity verification techniques [71], [72] should be employed before any analysis in order to handle attacks against the kernel protection technique.

The most comparable detection technique we could find were the proposals by Kruegel et al.[18], dAnubis [40], and Limbo [22]. Kruegel [18] is an static detection technique which models improper kernel memory access and based on the behavior observed classifies the driver as either legitimate or malicious. There are two issues with such approach. First the detection routine is limited to a very specific model and may miss rootkit drivers such as a spyware filter which is malicious but does not have improper kernel memory accesses. Furthermore, the proposed technique fails to differentiate between rootkits and their benign counterparts which may have similar behavior but different intent (i.e. one malicious and one legitimate). Lastly, the proposal by Kruegel et al. is implemented for the Linux operating system, unlike our proposed technique which operate in the Windows operating system.

The other two noted technique are based on dynamic analysis, although Limbo considers a few simple static features from the PE header of the file being analyzed. Unfortunately, and after a number of tries, we were unable to obtain proper access to dAnubis and Limbo for a comprehensive comparative analysis. Hence, we conducted a subjective comparison between these technique, which is presented in Table IV. dAnubis is designed as an analyzer with no framework for making a final judgment as to nature of the driver being analyzed. Moreover, it can not guarantee the execution of all parts of the driver in its simulator.

Limbo, on the other hand, employs a Bayesian classifier to label a given binary as either legitimate or malicious. However, many new behaviors of modern rootkits such as bootkits or hidden file systems are not considered and some assumptions made are no longer valid. For example, through our analysis of rootkits, we have observed that rootkit drivers try to imitate legitimate drivers by including symbol tables as debugging information. This is while Limbo consider this as a feature to be seen in legitimate drivers. In addition to the more comprehensive coverage in the proposed technique, including new types of rootkits (e.g. bootkits, HFS, and malicious filter drivers), the most important advantage of the proposed static detection technique is that it does not require a sandbox, emulator, or any isolated environment. Furthermore there are no concerns as to making sure that all parts of the code are executed with environment aware rootkits.

VI. CONCLUSION

In this manuscript we proposed a revisit to static analysis for detecting kernel-level rootkits. We made two important observations. First, one of the main approaches in penetrating the Windows kernel space is by employing kernel drivers. Second, there is usually little obfuscation applied to kernel-level codes and the possible obfuscation is much more prevalent in malicious drivers than legitimate ones. Based on these observations, we proposed a rootkits detection technique through static analysis of the kernel drivers. A big advantage of the static analysis approach, as opposed to dynamic analysis, is that it does not require the binary being analyzed to be executed.

Based on a number of observed trends, we proposed 50 features which are obtained from the dis-assembled driver. Furthermore, a large dataset consisting of 2200 malicious drivers and 2220 legitimate drivers where used to evaluate the effectiveness of the proposed features in distinguishing between legitimate and malicious drivers. Employing a C5 classifier, we were able to obtain and accuracy of 98.15% in classifying the malicious and legitimate drivers.

ACKNOWLEDGMENT

The authors would like to thank VirusShare [62] and specially J-Michael Roberts for their help in gathering the kernel drivers used in the experiments. Also VirusTotal's [73] collaboration in providing free academic access for verifying names of the sample files is appreciated. Lastly we would like to thank the anonymous reviewers for their constructive and detailed comments, with which we were able to improve this manuscript. This work was partially supported by ITRC under grant number 12188/500 (91/8/2).

REFERENCES

[1] A. Kapoor and R. Mathur, "Predicting the future of stealth attacks," in Virus Bulletin conference, 2011.
[2] (2012) Zegost - analysis of the chinese backdoor. [Online]. Available: http://artemonsecurity.blogspot.com/2012/12/zegost-analysis-of-chinese-backdoor.html
[3] P. Gutmann, "The commercial malware industry," in DEFCON conference, 2007.
[4] F. Op. (2008) The fu rootkit. [Online]. Available: http://www.hackerzvoice.net/ceh/CEHv6%20Module%2007%20System%20Hacking/FU_Rootkit/
[5] F.-S. Lab. (2005) Cut'n'paste rootkit-bots. [Online]. Available: http://www.f-secure.com/weblog/archives/00000559.html

[6] S. S. Response. (2005) W32.mytob.ar. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2005-041116-0718-99

[7] K. Kasslin, M. Ståhlberg, S. Larvala, and A. Tikkanen, "Hiden seek revisited–full stealth is back," in Proceedings of the 15th Virus Bulletin International Conference, 2005.

[8] J. S. Center. Ghost security suite ssdt hooks multiple local vulnerabilities. [Online]. Available: http://www.juniper.net/security/auto/vulnerabilities/vuln25709.html

[9] ——. Kaspersky internet security 6 ssdt hooks multiple local vulnerabilities. [Online]. Available: http://www.juniper.net/security/auto/vulnerabilities/vuln24491.html

[10] M. Russinovich. (2011) Using rootkits to defeat digital rights management. [Online]. Available: http://blogs.technet.com/b/markrussinovich/archive/2006/02/06/using-rootkits-to-defeat-digital-rights-management.aspx

[11] (2011) Returnil ssdt hooks listed as ¡unknown¿. [Online]. Available: http://www.wilderssecurity.com/showthread.php?t=303964

[12] V. Rusakov. (2011) Legit_bootkit. [Online]. Available: https://www.securelist.com/en/analysis/204792203/Legit_bootkits

[13] E. Rodionov. (2012) Win32/gapz: New bootkit technique. [Online]. Available: http://www.welivesecurity.com/2012/12/27/win32gapz-new-bootkit-technique/

[14] E. Rodionov and A. Matrosov, "Defeating anti-forensics in contemporary complex threats."

[15] S. Embleton, S. Sparks, and C. C. Zou, "Smm rootkit: a new breed of os independent malware," Security and Communication Networks, 2010.

[16] D. Zovi, "Hardware virtualization-based rootkits," Black Hat USA, 2006.

[17] S. Sparks, S. Embleton, and C. Zou, "Windows rootkits a game of hide and seek," Handbook of Security and Networks, p. 345, 2011.

[18] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in Computer Security Applications Conference, 2004. 20th Annual. IEEE, 2004, pp. 91–100.

[19] McAfee. (2011) Root out rootkits, an inside look at mcafee deep defender. [Online]. Available: hhttp://www.intel.ph/content/www/xa/en/enterprise-security/mcafee-deep-defender-deepsafe-rootkit-protection-paper.html

[20] S. Grobman et al., "Method and apparatus to detect kernel mode rootkit events through virtualization traps," Nov. 30 2010, uS Patent 7,845,009.

[21] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008, pp. 51–62.

[22] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in Recent Advances in Intrusion Detection. Springer, 2007, pp. 219–235.

[23] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in Proceedings of the 4th ACM European conference on Computer systems. ACM, 2009, pp. 47–60.

[24] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in Security and Privacy, 2006 IEEE Symposium on. IEEE, 2006, pp. 14–pp.

[25] J. Rutkowska, "Introducing blue pill," The official blog of the invisiblethings. org, vol. 22, 2006.

[26] ——, "System virginity verifier," in Hack In The Box Security Conference, 2005.

[27] J. Butler and G. Hoglund, "Vice–catch the hookers," Black Hat USA, vol. 61, 2004.

[28] M. LLC. Gmer rootkit detector. [Online]. Available: www.gmer.net

[29] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based file signatures for malware detection." in ICEIS (2), 2009, pp. 317–320.

[30] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. IEEE, 2001, pp. 38–49.

[31] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," Information Sciences, 2011.

[32] A.-D. Schmidt, J. H. Clausen, A. Camtepe, and S. Albayrak, "Detecting symbian os malware through static function call analysis," in Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on. IEEE, 2009, pp. 15–22.

[33] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining api calls," in Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010, pp. 1020–1025.

[34] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. IEEE, 2013, pp. 3422–3426.

[35] V. Moonsamy, R. Tian, and L. Batten, "Feature reduction to speed up malware classification," in Information security technology for applications. Springer, 2012, pp. 176–188.

[36] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011, pp. 309–320.

[37] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in Proceedings of the 20th international conference on World wide web. ACM, 2011, pp. 197–206.

[38] Z. Zhao, J. Wang, and C. Wang, "An unknown malware detection scheme based on the features of graph," Security and Communication Networks, vol. 6, no. 2, pp. 239–246, 2013.

[39] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 377–388.

[40] M. Neugschwandtner, C. Platzer, P. M. Comparetti, and U. Bayer, "Danubis–dynamic device driver analysis based on virtual machine introspection," in Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2010, pp. 41–60.

[41] B. Ulrich, M. Andreas, K. Christopher, and K. Engin, "Dynamic analysis of malicious code," in Journal in Computer Virology. Springer, 2006, pp. 67–77.

[42] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. IEEE, 2007, pp. 421–430.

[43] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in USENIX Security Symposium, 2007, pp. 275–290.

[44] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003, pp. 290–299.

[45] (2013) Direct rendering infrastructure wiki-nvidia. [Online]. Available: http://dri.freedesktop.org/wiki/NVIDIA/

[46] B. Schneier. (2007) Drm in windows vista. [Online]. Available: http://www.schneier.com/blog/archives/2007/02/drm_in_windows_1.html

[47] M. LLC. (2010) Khobe 8.0 earthquake for windows desktop security softwaree. [Online]. Available: http://www.matousec.com/info/articles/khobe-8.0-earthquake-for-windows-desktop-security-software.php

[48] G. Hoglund and J. Butler, Rootkits: subverting the Windows kernel. Addison-Wesley Professional, 2005.

[49] B. Blunden, The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Jones & Bartlett Publishers, 2012.

[50] M. Davis, S. Bodmer, and A. LeMasters, Hacking Exposed Malware and Rootkits. McGraw-Hill, Inc., 2009.

[51] R. Vieler, Professional rootkits. John Wiley & Sons, 2007.

[52] T. Klein, A bug hunter's diary. No Starch Press, 2011.

[53] W. Oney, Programming the Microsoft® Windows® Driver Model. Microsoft Press, 2010.

[54] P. Orwick and G. Smith, Developing drivers with the windows® driver foundation. Microsoft Press, 2010.

[55] Osr online forums. [Online]. Available: http://www.osronline.com/page.cfm?name=listserver

[56] A. Srivastava and J. Giffin, "Automatic discovery of parasitic malware," in Recent Advances in Intrusion Detection. Springer, 2010, pp. 97–117.

[57] G. Kroah-Hartman, "Things you should never do in the kernel," Linux Journal, vol. 2005, no. 133, p. 9, 2005.

[58] ErikMouw. (2006) Why writing files from the kernel is bad ? [Online]. Available: http://kernelnewbies.org/FAQ/WhyWritingFilesFromKernelIsBad

[59] Microsoft. (1999) How to open a file from a kernel mode device driver and how to read from or write to the file. [Online]. Available: http://support.microsoft.com/kb/891805

[60] ——. How to modify executable code in memory. [Online]. Available: http://support.microsoft.com/kb/127904

[61] K. Kasslin. (2010) Evolution of kernel-mode malware. [Online]. Available: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm
[62] M. LLC. Virusshare malware repository. [Online]. Available: virusshare.com/
[63] D. French, "Fuzzy hashing techniques in applied malware analysis," 2011.
[64] Hex-rays. Ida overview. [Online]. Available: https://www.hex-rays.com/products/ida/overview.shtml
[65] [Online]. Available: http://sharif.edu/~kharrazi/rtk-stat.php
[66] Zwmapviewofsection routine. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff566481%28v=vs.85%29.aspx
[67] F. Factories. (1999) MS Windows NT kernel description. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff536385%28v=vs.85%29.asp
[68] "Pearson r-correlation coefficient," http://www.strath.ac.uk/aer/materials/4dataanalysisineducationalresearch/unit4/pearsonr-correlationcoefficient/.
[69] E.Rodionov and A.Matrosov. (2011) King of spam:festi botnet analysis. [Online]. Available: www.welivesecurity.com/wp.../king-of-spam-festi-botnet-analysis.pdf
[70] T. Butler, "The cat-and-mouse game: The story of malwarebytes chameleon," http://blog.malwarebytes.org/intelligence/2012/04/the-cat-and-mouse-game-the-story-of-malwarebytes-chameleon/, 2013.
[71] "Osiris: Host integrity management tool," http://www.bnl.gov/itd/unix/administration/osiris.asp.
[72] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in Proceedings of the 2nd ACM Conference on Computer and Communications Security. ACM, 1994, pp. 18–29.
[73] Virustoal free online virus, malware and url scanner. [Online]. Available: https://www.virustotal.com/

S. Atefeh Musavi received her B.S. degree in computer engineering from the Amirkabir University of Technology, Tehran, Iran in 2011 and her M.S. degree in information technology from the Sharif University of Technology, Tehran, Iran in 2013. Her research interests include Malware detection, OS security, and information forensics.

Mehdi Kharrazi received his B.E. degree in electrical engineering from the City College of New York and his M.S. and Ph.D. degrees in electrical engineering from the Department of Electrical and Computer Engineering, Polytechnic University, Brooklyn, New York, in 2002 and 2006 respectively. He is currently an Assistant Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His current research interests include computer and network security.