# Detecting Stealthy Malware with Inter-Structure and Imported Signatures

Bin Liang[1,2] Wei You[1,2] Wenchang Shi[1,2] Zhaohui Liang[1,2]
[1]Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, Beijing 100872, P.R. China
[2]School of Information, Renmin University of China, Beijing 100872, P.R.China
{liangb, youwei, wenchang, lzh}@ruc.edu.cn

## ABSTRACT

Recent years have witnessed an increasing threat from kernel rootkits. A common feature of such attack is hiding malicious objects to conceal their presence, including processes, sockets, and kernel modules. Scanning memory with object signatures to detect the stealthy rootkit has been proven to be a powerful approach only when it is hard for adversaries to evade. However, it is difficult, if not impossible, to select fields from a single data structure as robust signatures with traditional techniques. In this paper, we propose the concepts of inter-structure signature and imported signature, and present techniques to detect stealthy malware based on these concepts. The key idea is to use cross-reference relationships of multiple data structures as signatures to detect stealthy malware, and to import some extra information into regions attached to target data structures as signatures. We have inferred four invariants as signatures to detect hidden processes, sockets, and kernel modules in Linux respectively and implemented a prototype detection system called DeepScanner. Meanwhile, we have also developed a hypervisor-based monitor to protect imported signatures. Our experimental result shows that our DeepScanner can effectively and efficiently detect stealthy objects hidden by seven real-world rootkits without any false positives and false negatives, and an adversary can hardly evade DeepScanner if he/she does not break the normal functions of target objects and the system.

## Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection

## General Terms

Design, Security

## Keywords

Rootkit, Signature, Malware Detection

## 1. INTRODUCTION

Kernel mode rootkits have been proven to be a significant threat to computer security. They hide some system objects such as processes, sockets and kernel modules to conceal their presence on a victim host. The usual stealth techniques employed by rootkits include Kernel Object Hooking (KOH) and Direct Kernel Object Manipulation (DKOM). KOH rootkits hijack kernel control flow while DKOM rootkits directly modify kernel data objects. For example, the *hp* rootkit unlinks the target process from the task list maintained by Linux kernel to prevent it from being discovered with utility *ps*.

One way to detect hidden objects is to scan kernel memory with signatures to seek the data structures of processes, sockets, and kernel modules, and compare an object list inferred from these data structures with the output of the system standard utility tools, such as *ps*, *netstat*, and *lsmod*. When an object is found in the scan results but not in the output of utility tools, it means that a possible stealthy malware is detected. Some detection tools and forensic analysis tools have been implemented to scan kernel memory using signatures to detect hidden objects [3][8][9][21][23]. Signature-based scanning can be used in online and offline forensic.

The signature-based scan method can only be effective if it is difficult for adversaries to evade. Unfortunately, some signatures employed by detection tools can easily be evaded by modifying some fields of object data structures. For example, a signature used by KSTAT [3] to detect hidden kernel modules is that the value of the first 4 bytes of the target memory area (the *size_of_struct* field of *module* structure) is equal to the size of the *module* data structure. Adversaries can set the value of *size_of_struct* field to another value (e.g., setting it to zero), to conceal rootkit LKM (Loadable Kernel Module) from KSTAT detection. Furthermore, the value of this field is irrelevant to the operations of LKMs, the code and data of a hidden kernel module can be still correctly accessed after modifying *size_of_struct* field.

A natural question is that which fields of a target data structure can be used as scanning signatures to effectively detect stealthy malware and difficult to be evaded. The fields chosen as signatures should correlate closely to the system operations. Any attempting to modify these values will lead functions of objects to fail or the operating system to crash. Thus, adversaries will not want to modify these fields to avoid losing the control of hidden objects or the victim system. Take the processes as an example, process hiding is a common feature of kernel rootkits, a kernel rootkit may be employed to hide the presence of a wide variety of user-space malware. With regards to this, Dolan-Gavitt et al. [12] proposed an automated fuzzing technique for generating robust signatures for kernel data structures of processes in Windows. Some constraints are found for robust signatures in the EPROCESS data structure. They can be used by a scanner to detect hidden processes. However, there still are two obstacles to obtaining robust signatures for some security-critical kernel data structures.

First, the fields relevant to the correct operations of a target kernel data structure may not be able to act as signatures because it is difficult to distinguish the target kernel structure from memory based on these fields. For example, the *socket* data structure in Linux kernel has several pointer fields, such as *ops* and *file*. Modifying these fields will result in communication failure of sockets. But these fields point to some dynamic memory areas rather than hold specific constants. It is impossible to accurately find *socket* structures by scanning based on these fields.

Second, all fields of structures of some *passive* kernel objects are hardly relevant to system operations. The operations of system and objects still are normal even after modifying them arbitrarily. For example, the main purpose of the Linux kernel module data structure is providing some operation handles and information for unloading LKM. According to our experiments, the codes imported by an LKM still execute correctly even after setting all fields of its module structure to zero or any other arbitrary value.

An elaborate scanner should/can discover the hidden sockets and kernel modules and resist possible evasion techniques. The above discussion shows that it is difficult to use fields limited in a single kernel data structure (*intra-structure*) as scan signatures to detect hidden objects when facing skilled adversaries. However, traditional approaches generate signatures commonly from the intra-structure view and cannot provide effective detection capability to some critical kernel objects, such as sockets and kernel modules.

To address the above challenges, we propose the concepts of *inter-structure* signature and *imported signature* and present an approach to detect stealthy malware based on them. The key idea is to use the cross-reference relationships of target data structure and other related data structures as signatures to detect the hidden objects mentioned in the first obstacle, and to import some extra information into regions attached to target data structures as signatures to detect the hidden objects mentioned in the second obstacle. According to these concepts, we can either infer some invariants from multiple related data structures or introduce new invariants as robust signatures. With these signatures, a scanner can effectively detect some stealthy malware that can hardly be discovered with prior techniques. In this paper, we construct four invariants to detect hidden process, socket, and kernel module in Linux. One of them is derived from some imported information in the text section of kernel module to reverse search module structures; the other three reflect some cross-reference relationships of *task_struct*, *socket*, and related data structures to recognize process and socket objects respectively.

It is important to note that the imported signatures should be protected to prevent attacker modifying them to evade scanning. The hardware only provides page-level protection. But Linux doesn't page-align the sections of a kernel module and cannot set appropriate page access permissions to them when loading modules. Consequently, the text section of a kernel module is still writeable. More seriously, even after setting the text section to read-only, a kernel mode rootkit can reset it to writeable and modify its content. To protect the imported signatures, we add a new page, called *signature page*, to store the tag data and protect it by a monitor in a virtual machine hypervisor XEN [7]. The signature page will be allocated as the first page of text section and be set to read-only. Any attempts to modify its access permissions will be trapped into hypervisor, and the monitor will deny the requests to set access permissions of signature pages.

A prototype system called DeepScanner is implemented to detect stealthy malware in Linux. The essential function of DeepScanner is enumerating all processes, sockets and kernel modules in the system by using above four invariants as signatures to scan kernel memory. Our experiments show that DeepScanner can effectively detect all stealthy processes, sockets, and kernel modules hidden by real-world rootkits without false positives or false negatives. Besides, we also implement an experiment rootkit to imitate evasion attacks by modifying the fields related to signatures. Our works demonstrate that rootkits cannot evade DeepScanner without breaking the normal functions of target objects and the operating system.

The rest of this paper is organized as follows. Section 2 describes related works. Section 3 presents the concepts of inter-structure and imported signatures. Section 4 describes the implementation of prototype system. Section 5 presents the experiments. Section 6 discusses the limitations and future work, and Section 7 concludes this paper.

## 2. RELATED WORKS

Recently, a large number of studies pay much attention to the detection of rootkits. Since almost all rootkits possess the nature of hiding themselves, it is a reasonable way to detect them by digging the hidden objects. The cross-view based detection is a simple but valid technique which detects hidden entities via comparing the differences between *untrusted view* collecting from high-level and *trusted view* collecting from low-level. This notion is initially proposed by Wang et al. in their Strider GhostBuster system [25].

The key factor and primary difficulty of this method is the way to get the trusted view. Several approaches have been proposed to address it. For example, Klister [20] collects all existing processes from scheduler's thread list rather than the system-wide process list. Petroni et al. [17] put forward an architecture which check the semantic integrity violations between the All-Tasks linked-list and Running-Tasks linked-list in Linux. But unfortunately, an evasion for this kind of detection has been demonstrated, which bypasses detection via replacing the OS scheduler with a modified copy [1].

Another attempt to obtain the trusted view is memory searching. Due to the fact that every entity has its correlative kernel data structure maintained by operating system which indicates its existence, it is a feasible way to find all existing entities via locating them in memory with the features of specific data structures. Such kinds of kernel memory scanning tools have been proposed by some researchers [3][8][9][21][23]. However, the signatures used by these tools are almost brittle and non-essential that making them easy to be evaded, even by simply modifying some bits of the data structure without preventing the entity from working correctly [24].

Dolan-Gavitt et al. [12] developed a novel method for systematically selecting features from a kernel data structure that can be used to create robust signatures with the support of fuzzing technique. Attempts to evade this kind of signatures by modifying the data structure contents will cause the OS to consider the object invalid. Their efforts show that signature based memory scanning is an effective and unfailing way, and it is possible to find out some robust signatures that are infeasible for attackers to modify. However, selecting candidate signatures is limited in a single data structure in their method, such as "value equals X" or "value is between X and Y". Consequently, this method cannot discover the inter-structure features as signatures which are proven to be indispensable to detect some objects by our research. Based on our observations and experiments in section 3.1, we can find that it is impossible to generate a feasible robust intra-structure signature for some important system objects, such as socket objects in Linux kernel.

Carbone et al. proposed KOP [10] which involves building a global points-to graph for kernel memory mapping and kernel integrity checking. The output of KOP's memory analysis component is an object graph whose nodes are instances of objects in the memory snapshot and edges are the pointers connecting these objects. Given an object type (e.g., process), a corresponding trusted view can be obtained from the list of all the objects of that type found by KOP in a memory snapshot. However, this method requires that data structure instances be reachable starting from the root(s) of the graph. The path from root(s) to target objects may be cut by adversaries without breaking the normal functions of target objects. In a latest research, Lin et al. developed a novel approach [16], called SigGraph, to generate graph-based signatures by a data structure definition extractor and a dynamic profile. Unlike KOP, SigGraph does not require the object reachability and hence supports brute force memory scanning that can start at any kernel memory address. These two methods can automatically derive some useful information for detecting malwares by employing static analysis technique. Compared with them, our method is lightweight and gives special consideration to preventing possible evasion attacks.

Baliga et al. [6] presented an approach to automatically detect rootkits and implemented a tool Gibraltar. Their key idea is to externally observe the execution of the kernel during a training period and hypothesize invariants on kernel data structures. These invariants are used as specifications of data structure during an enforcement phase; violation of these invariants indicates the presence of a rootkit. However, this approach doesn't take into account some important attack methods of rootkits, such as hiding sockets and kernel modules. Especially, due to the function-independence of fields in the kernel module data structure (will be demonstrated in section 3.2), it may be impossible to infer an invariant for kernel modules by observing the execution of the kernel.

HookFinder [28] and HookMap [27] focus on identifying and extracting hooks placed by rootkits. HookFinder performs impact analysis using dynamic tainting to identify hooks placed by a rootkit in the kernel execution paths. HookMap uses a more elaborate method of identifying all potential hooks in the execution path of kernel code that is induced by the execution of Linux utility programs such as *ls*, *ps*, and *netstat* in RedHat Fedora Core 5. HookMap

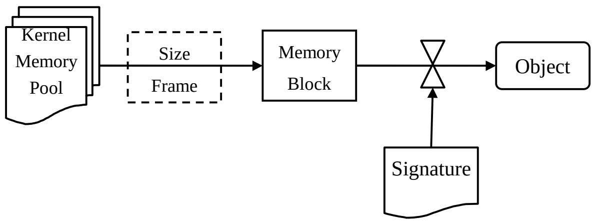


**Figure 1: Signature-based scanning.**

found that there exist 35 kernel hooks in the kernel-side execution path of *ls* that can be potentially hijacked for manipulation. Similarly, there are 85 kernel hooks for *ps* and 51 kernel hooks for *netstat*, which can be respectively hooked for hiding processes and network activities. It is tedious to detect potential malicious attacks by monitoring so many hook points.

File-signature based detection is a classical method in the area of virus detection, which has repeatedly been proposed to identify particular classes of security threats. Some rootkit detectors [2][5] employ this method to scan system files for a sequence of bytes that comprise a fingerprint which is unique to a particular rootkit. This method is simple and effective. But due to its dependence on the prior knowledge of the known malware, it lacks of the ability to defend against unknown or mutative malwares.

Another area of related work is using virtual machine monitor (VMM) to provide a high resistance to attacks from inside the system. The advantage of VMM-based detection is that it has the higher privilege over kernel-targeted malwares, which the traditional host-based anti-malware systems limit in. Since Garfinkel et al. [13] introduced the VMM-based intrusion detection system and VM introspection technique, it is widely adopted by researchers to detect and analyze intrusions. In some works [19][22], VMM is used to guarantee the integrity of executed kernel code. This prevention technique monitors the execution of kernel code in the hypervisor level, and only allows the authorized code to be run in the kernel's address space. Besides, some researches introduce the cross-view based detection into hypervisor level. VMwatcher [14] captures a process when the control-register CR3 changes, that means it could only discover the process at the moment it enters running state, and could not gain the information of all the existing processes at a time. Lycosid [15] uses statistical inference techniques to obtain the trusted view. Due to the limitation of statistical method, the low-level view Lycosid gets is just more reliable but not accurate.

VMM technique is also used to protect some essential kernel data. Rhee et al. [18] implemented a system to monitor kernel memory access using VMM-based policies in QEMU [4], adopting memory access control in the hypervisor level to protect the essential kernel data. Wang et al. [26] relocated some important kernel hooks to a dedicated page-aligned memory space and then regulate accesses to them with a monitor in XEN hypervisor. In this paper, we also employ VMM technique to protect the integrity of imported signatures.

## 3. METHODOLOGY

The process of the signature-based scanning method is shown in Figure 1. In the first place, a scanner fetches a block of memory from kernel memory pool according to a size frame of target object data structure. Then, the scanner will check whether the content of the memory block matches a signature. If so, the memory block will be regarded as a desired object data structure, and the information about the corresponding object can be extracted from it.

To reach the goal, scanning signatures must meet the following two requirements:

1) They must have enough discriminating power to distinguish object data structures without high false positives.

2) It is impossible for adversaries to evade them without breaking the functions of target objects and operating system.

An effective signature must meet both of the above two requirements. The fields that only meet one of them cannot act as an effective signature. For example, there are many pointer fields in Linux kernel data structures; they point to some operation handles or other data structures. These fields are commonly closely related to the functions of their host objects. Attempting to modify them will break the normal operations of the host objects or crash the whole system. However, the values of these pointers are variable with regards to different object states or kernel versions. The only invariant can be deduced from them is that the values of these pointers are kernel-space addresses. Based on the property, a scanner cannot obtain desired data structures without high false positives. On the contrary, the type, size, and state fields of kernel data structures are often comparatively static. For example, the state field of the Linux process descriptor (*task_struct*) describes the current condition of a process, and it may possesses one of five possible flags (*TASK_INTERRUPTIBLE*, *TASK_UNINTERRUPTIBLE*, *TASK_RUNNING*, *TASK_STOPPED* and *TASK_ZOMBIE*) to indicate the process's state. As another example, the *size_of_struct* field of Linux module structure holds the size of the data structure. The invariants deduced from these fields can effectively identify specific kernel data structures. But, adversaries can modify the values of these fields to evade scanning without affecting the normal operations of system.

In the rest of this section, we take Linux kernel 2.6 as example to describe two new kinds of signatures and give four invariants to detect stealthy malware. Because some real-world rootkits employ version-dependent attack techniques, they cannot work in the newest version kernel. Without loss of generality, we choose kernel 2.6.9 (Redhat AS4) as the target platform.

## 3.1 Inter-structure Signatures

Not every kernel data structure has ideal fields meeting the above two requirements at the same time. To some kernel data structures, we cannot deduce effective signatures from them directly. To this end, we try discover some cross-reference relationships to generate robust signatures.

### *3.1.1 Sockets*

Sockets are a common target for attackers to hide to conceal malicious communications. In Linux, the *socket* data structure is used to describe network socket objects. As

```
struct socket
{
        socket_state            state;
        unsigned long           flags;
        struct proto_ops        *ops;
        struct fasync_struct    *fasync_list;
        struct file             *file;
        struct sock             *sk;
        wait_queue_head_t       wait;
        short                   type;
        unsigned char           passcred;
};
```

**(a)**

```
struct socket_alloc
{
     struct socket socket;
     struct inode vfs_inode;
};
```

**(b)**

**Figure 2: Linux *socket* and *socket_alloc* structure.**

shown in Figure 2 (a), the *socket* data structure of Linux kernel 2.6.9 is very simple. It consists of nine fields. Among them, *state*, *flags*, *type*, and *passcred* fields meet the first requirement but do not meet the second one. We can do a simple experiment to demonstrate this. An experimental LKM is designed to clear these fields of a connected SSH socket data structure. After clearing them, we can still access a remote server correctly via the SSH connection. On the contrary, *ops*, *fasync_list*, *file*, *sk*, and *wait* fields meet the second requirement but not the first.

According to the above analysis, we can see that no appropriate fields can act as signatures in the *socket* data structure. In other words, it is impossible to generate an effective signature from a single data structure (intra-structure) for scanning socket objects. We need to extend the scope of choice to find an effective signature.

In Linux kernel, there are many pointer fields in kernel data structures, which point to some operation handles or other data structures. When a pointer field of a data structure points to a related data structure; a pointer field of the related data structure may also point to the source data structure directly or indirectly. These fields construct a kind of cross-reference relationship of related data structures. If the cross-reference relationship reflects an invariant about target objects that satisfies both of the two requirements, it can be used as an effective signature. Compared with the signatures from fields in a single data structure, this kind of signature involves multiple related data structures. We call it as *inter-structure signature.*

As shown in Figure 2 (b), in Linux kernel 2.6.9, a *socket* data structure exists as a field of *socket_alloc* data structure. The *socket_alloc* data structure is an item of a cache that stores pairs of socket and its virtual file system inode (*vfs_inode*).

With regard to the *socket* data structure, its *ops*, *file*, *sk*, *fasync_list*, and *wait* fields are used to point to some related data structures. Among them, the *file* field points to a file descriptor (*file*). The file descriptor is an interface for applications to manipulate sockets. In a *file* data structure, a pointer field *f_dentry* points to a *dentry* data structure. Subsequently, a field *d_inode* of the *dentry* data structure

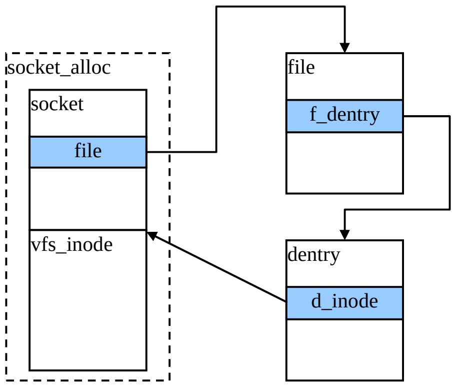


**Figure 3: A relationship of multiple socket related data structures.**

> **Invariant 1**: for a *socket_alloc* data structure *sa*:
>
> (*sa.socket*)->*file*->*f_dentry*->*d_inode* points to *sa.vfs_inode*

**Figure 4: Invariant for the *socket_alloc* data structure.**

should point to the *vfs_inode* field of a *socket_alloc* data structure that consists of the source socket data structure and its virtual file system inode. The relationship of above data structures is shown in Figure 3. An invariant shown in Figure 4 can be inferred as a signature from the relationship.

Using the above invariant, we can enumerate all socket objects in a target Linux system. At the first step, a scanner will search kernel cache memory to get all *socket_alloc* data structures based on the invariant. Then, a *socket* data structure can be easily collected as a field of the *socket_alloc* data structure.

Although we take the *socket* data structure as an example to introduce inter-structure signatures, we are sure that the concept can be applied to other data structures.

### *3.1.2 Processes*

In Linux kernel, the *task_struct* data structure represents a process of system. The *task_struct* data structure is a relatively complex data structure and contains many fields.

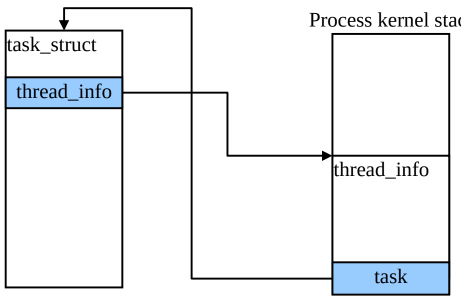


**Figure 5: A relationship between the *task_struct* and *thread_info* data structures.**

> **Invariant 2**: for a *task_struct* data structure *t*:
>
> (*t.thread_info*)->*task* points to *t*
>
> Or for a *thread_info* data structure *th*:
>
> (*th.task*)->*thread_info* points to *th*

**Figure 6: Invariant for the *task_struct* and *thread_info* data structures.**

> **Invariant 3**: for each *socket_alloc* data structure *sa*, there exist a *task_struct* data structure *t* and a *file* data structure *f*, such that:
>
> - *f* ∈ (*t.files*)->*fd_array*, and
> - (*f.f_dentry*)->*d_inode* points to *sa.vfs_inode*, and
> - (*sa.socket*)->*file* points to *f*

**Figure 7: Invariant for the *socket_alloc* data structure based on the relationship between processes and sockets.**

However, there are only a few fields can be used to identify it, such as *state* and *pids*. Unfortunately, these fields do not meet the second requirement, i.e., they can be modified arbitrarily without breaking the operations of the target process. We design an experimental LKM to modify these field of a *task_struct* data structure to invalid values. After modifying them, the target process still run correctly.

Similar to socket objects, the process objects can be explored using an inter-structure signature. As shown in Figure 5, there is a *thread_info* data structure stored in the kernel space stack of the process in Linux kernel 2.6.9; the *task* field of the data structure is a pointer to the process's actual *task_struct* data structure. At the same time, the *thread_info* pointer field of a *task_struct* data structure points to its *thread_info* data structure conversely. The cross-reference relationship is closely related to the operations of processes; breaking it will lead to system crash. On the other hand, using the relationship can effectively identify *task_struct* data structures. Thus, we can infer an invariant shown in Figure 6 for scanning process objects.

During scanning, scanner can fetch all possible process kernel stack memory block and check whether its end part matches the invariant about *thread_info*. If it does, the corresponding *taks_struct* data structure can be obtained from its *task* pointer.

After getting all *task_struct* data structures, there is another way to deduce all *socket* data structures. In Linux kernel, a socket usually belongs to a process as an open file. The *files* field of *task_struct* is a pointer to a *struct_files* data structure used to store open files information. We can reference an open file via a file descriptor array (*fd_array*) in *struct_files*. Based on Invariant 1, we can get another invariant about socket objects.

A scanner can collect all open file descriptors from obtained *task_struct* list and check one by one using Invariant 3. If a *socket_alloc* data structure can be deduced from a file descriptor, the scanner will find a possible socket object.

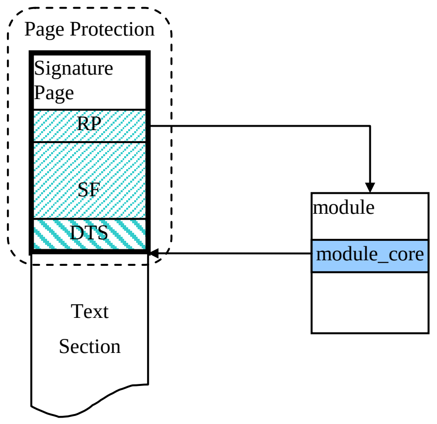


**Figure 8: An imported signature for the *module* data structure.**

> **Invariant 4**: for each *module* data structure *m*, there exists a signature page *p*, such that:
>
> - *p* contains a DTS, and
> - *p.RP* points to *m*

**Figure 9: Invariant for the *module* data structure.**

## 3.2 Imported Signatures

For some objects, all fields of their data structures are hardly relevant to their functions or operations of the whole system. The related operations still are normal even after all fields are modified arbitrarily.

In general, kernel mode rootkits are implemented as kernel modules (e.g., a device driver) and loaded into kernel space of operating system in various ways. The most important task of kernel mode rootkits is to hide themselves. To this end, an effective scanner should be able to discover hidden kernel modules. In Linux, the kernel module data structure is used to represent a loaded LKM. Unfortunately, all fields of the module data structure are independent of its LKM function except unloading. After a module being loaded, the values of all fields of its data structure are insignificant to executing its code or accessing its data.

We do an experiment to demonstrate the function-indepen dence of fields of the *module* data structure. First, we load an LKM which contains two functions: *mygetsid* () and *modifymyself* () and a char array *mystring*. The *mygetsid* function will hook the *getsid* system call and output *mystring* to system log; the *modifymyself* function is used to set the memory block of the *module* data structure to zero. Second, we will call the *modifymyself* function to clear the *module* data structure. Final, we will call *getsid* system call from a user space application to trigger the *mygetsid* function. We can examine system log to determine whether the code and data of kernel module can be executed or accessed correctly after related *module* data structure being cleared. In the experiment, we can observe that the correct contents of *mystring* are output by *mygetsid* function to system log. The result of the experiment proves that modifying any field of a *module* data structure cannot break the normal functions of corresponding LKM.

According to the above experiment results, it is confirmed that no fields of the *module* data structure can be used as either an effective signature or a part of an inter-structure signature. To detect hidden LKMs in Linux, we need introduce some extra information into memory regions attached to *module* structures as an effective signature. We call this kind of extra information as *imported tag*.

Adding some distinctive data in the *module* data structure is a natural and direct way to import signatures. However, this way is not robust enough to counter evasion attacks. Essentially, the imported tags are the same with those original fields from the viewpoint of attackers. Adversaries can trivially evade this kind of signatures by modifying the imported tags to some malformed values. Especially, a kernel mode rootkit, as a part of OS kernel, has enough power to do this. A possible improvement of this way is relating the imported tags with some elementary functions of LKMs, e.g., calling the code provided by LKM. As results, adversaries tampering with the imported tags will break the elementary functions of LKMs or lead to system crash. But it is very troublesome to change the operation logic of the operating system kernel. Besides, the change to kernel is platform-dependent.

A more feasible choice is to protect the imported tags from being modified. Unfortunately, this kind of protection requires byte-level granularity while modern hardware platforms can only provide page-level protection. The tags in a data structure will be co-located together with original fields that may be written during the lifecycle of target objects. Because they may be located in the same page, it is improper and unadvisable to protect imported tags and writable original fields with the same access control permission (read only).

To bridge the protection granularity gap, we can place the imported tags to a dedicated page-aligned memory area and co-locate them together with some data of the same nature. Consequently, the imported tags can be protected from being tampered based on hardware protection mechanism. Some identification information can be introduced in the tags to identify and refer target objects, and can be used as an effective signature. We call these identification information as *imported signature*.

In Linux kernel, the code and data of an LKM will be loaded into some kernel space memory. They can be referenced using the *module_core* field of the *module* data structure. The text section of LKM is an optional place to store the imported tags. It is a natural way to protect imported tags and the code of LKM from being written. But it will incur a code relocation problem due to the effect to the layout of the code. To avoid the troublesome code relocating, we add a new page prior to the text section to place the imported tags when loading an LKM. This page will be set to read-only to avoid malicious tampering. We call the page as *signature page*. As shown in Figure 8, a distinctive tag for scanning (DTS) and a pointer to reference corresponding module data structure (RP) are stored in the page. Besides, some static fields of the *module* data structure (SF) are also stored to check the integrity of the data structure. Accordingly, we can infer an invariant shown in Figure 9 for scanning module objects.

Scanner can traverse the kernel page memory to get the information via checking whether there is a DTS in current memory page. A page containing DTS will be regarded as a

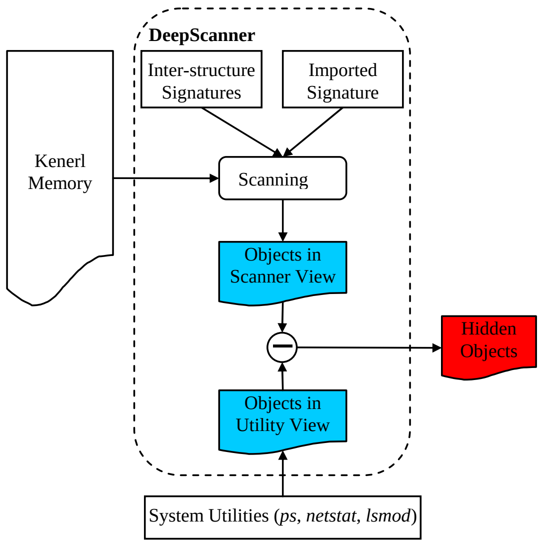


**Figure 10: An imported signature for the *module* data structure.**

signature page, which corresponds to a *module* data structure. Scanner can get the information about the module via the RP pointer in the signature page and further check the integrity of the of the module with SF.

### 3.3 Signatures for Linux 2.4

Because some rootkits are aimed at Linux 2.4, we also construct corresponding invariants for Linux 2.4.18.

The signatures for Linux 2.4.18 are similar to the above signatures. But for Linux 2.4.18, the *task_struct* data structures are directly allocated in the process stack and there is not a *thread_info* data structure in kernel. For this reason, we use an imported signature to detect hidden processes on Linux 2.4.18 platform. The related invariant is similar to Invariant 4.

## 4. IMPLEMENTATION

To evaluate the usefulness of our approach, we implemented a prototype system DeepScanner to detect the stealthy malware in Linux. The main purpose of DeepScanner is to demonstrate the effectiveness of inter-structure and imported signatures. For the sake of convenience, DeepScanner is currently implemented as an LKM and a user mode GUI console. DeepScanner LKM is responsible for scanning kernel memory and reporting results to the console. The console is used to manage scanning operations, collect various results, and make cross-view comparison.

As shown in Figure 10, DeepScanner uses cross-view technique to detect the hidden objects. First, DeepScanner traverses target kernel memory and enumerating all processes, sockets and modules according to above 4 invariants. Second, it will collect the output of system utilities, including *ps*, *netstat*, and *lsmod*. Finally, DeepScanner will compare the two results of different views. If an object is found in the scanner view but not in the utility view, it means a possible stealthy malware is detected. Besides, because the imported signature is the base to detect hidden modules, we also implement a hypervisor based monitor to protect the signature page.

In the rest of this section, we will describe three kinds of objects detection in detail based on Linux 2.6.9 platform.

### 4.1 Hidden Sockets

Socket hiding is a common goal for attackers to conceal their malicious communications. DeepScanner can detect hidden sockets based on Invariant 1 or Invariant 3.

When users only want to detect hidden sockets, DeepScanner uses Invariant 1 as scanning signature. In Linux 2.6.9, the *socket_alloc* data structure is stored in a dedicated cache allocated by calling kernel routine *kmem_cache_alloc*. To speed up scanning, DeepScanner employs a directive scanning strategy. It fetches candidate memory block by traversing the kernel slab list with specific item size. DeepScanner assume each candidate block *cb* as a *socket_alloc* data structure and check whether it satisfies Invariant 1, namely check whether $((struct\ socket\_alloc)(cb)).vfs\_inode$ is pointed to by $(((struct\ socket\_alloc)(cb)).socket)\text{->}file\text{->}f\_dentry\text{->}d\_inode$. If it meets Invariant 1, $((struct\ socket\_alloc)(cb)).socket$ will be the desired *socket* data structure.

On the other hand, if DeepScanner already have a process descriptor list, it can easily enumerate all *socket* data structures via traversing the *files* field of each *task_struct* data structure according to Invariant 3.

### 4.2 Hidden Processes

Process hiding is often the primary feature of a kernel mode rootkit. DeepScanner can detect hidden processes based on Invariant 2. In addition, given a complete processes list, DeepScanner can further get all sockets using Invariant 3.

According to Invariant 2, there are two scanning modes to choose: scan *task_struct* directly or scan *thread_info* and then get *task_struct* indirectly. To increase scanning speed, DeepScanner choose the latter. Prior to the 2.6 kernel series, the *task_struct* is stored at the end of the kernel stack of each process. In kernel 2.6, the *task_struct* is moved from process kernel stack to a *slab* block. Instead, the *thread_info* is stored at the bottom of the kernel stack. On x86 platform, the process kernel stack is 8kb-aligned (two pages), so the address of the *thread_info* data structure is also 8kb-align. Based on this fact, DeepScanner check candidate memory block 8kb by 8kb rather than byte by byte, namely one scanning step of DeepScanner is 8kb. During detecting hidden processes, DeepScanner can only fetches the memory blocks begin with 8kb-aligned address, and check whether they satisfy Invariant 2. This scanning mode dramatically speeds up scanning. As a comparison, if scan *task_struct* directly, the scanner may need to traverse target memory space byte by byte.

Given a *thread_info* structure, getting the corresponding *task_struct* is very simple. It can be obtained by referencing the memory area pointed by *thread_info.task*. The information in *task_struct* data structures obtained by scanning will construct a scanner view of system processes.

### 4.3 Hidden Kernel Modules

DeepScanner uses imported signatures to detect hidden kernel modules. In order to introduce imported signatures, we patch the module loading mechanism of Linux kernel. During module loading, the patched kernel will add a new page prior to the text section of module. The page will be allocated in HIGHMEM region of kernel memory. Deep-

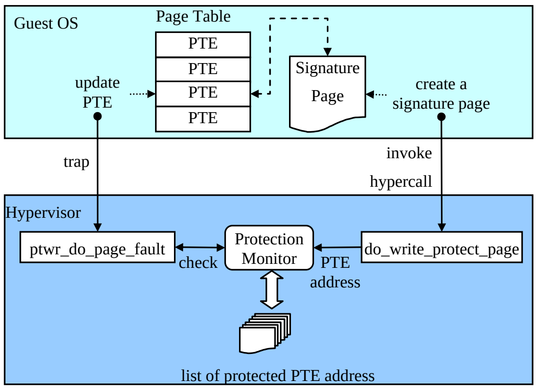


**Figure 11: Signature Page Protection in XEN Paravirtualization Mode.**

Scanner check each page in HIGHMEM region to check whether it contains a DTS. The DTS is an 8 bytes sequence 0x89c089db89c989d2 corresponding to a NOP instructions segment (*mov eax, eax*; *mov ebx, ebx*; *mov ecx, ecx*; *mov edx, edx*). The NOP instruction sequence has enough discriminability and normally doesn't exist in the system memory space according to our observations. Furthermore, it is fully side-effect-free. According to our experiments, the sequence can be used to unambiguously identify signature pages.

The RP part of a signature page is a pointer to corresponding *module* data structure, and the SF part stores some static fields of the *module* data structure, including *name*, *module_core*, *syms*, etc. After getting a signature page, DeepScanner will get necessary information of a *module* data structure via RP pointer and further check the integrity of the data structure via comparing it with SF.

A problem need to be considered is that attackers can introduce noise by constructing some malformed imported signature pages following our design, which may lead to false positives. From the viewpoint of system security detection, it is sure that the integrity of system has been compromised when a scanner found a malformed signature. In other words, the system has been intruded by attackers. Because a primary goal of attackers is concealing their behaviors, we think attackers don't want to interfere security detection mechanism by this way.

To DeepScanner, signature page protection is critical to discover hidden kernel modules. It will be describe in following subsection.

## 4.4 Signature Page Protection

The integrity of signature pages determines the effectiveness of hidden kernel module detection. It is not enough for signature pages protection just only to set the signature page to read-only in its page table entry (PTE). Kernel mode rootkits can easily modify the PTE and set the signature page to writable.

In this paper, we employ VMM technique to protect the integrity of imported signature pages. We implement a monitor in XEN hypervisor. Any attempts to modify the access permissions of signature page will be trapped into the hypervisor. The monitor will terminate the operation to modify the PTE of signature pages.

In XEN para-virtualization mode, a guest OS is not permitted to modify its page tables directly, any update of page tables will be trapped into hypervisor [11]. The function *ptwr_do_page_fault* will be called to handle the update operation when guest OS is in kernel mode. Based on the model, we implement a protection monitor (PM) in XEN hypervisor. It will intercept *ptwr_do_page_fault* to check if the target PTE to be updated to writeable corresponds to a signature page. If so, the update operations will be terminated by PM. As shown in Figure 11, to identify the signature pages in hypervisor, we add a new hypercall *do_write_protect_page* to XEN. When a new signature page is created, system will invoke *do_write_protect_page* to notify PM and send the virtual address of the related PTE to it. The PM maintains a list to store the addresses of PTEs need to be protected. With the support of the list, the PM can know which update to PTE needs to be checked. Consequently, the signature pages always are read only after being created.

In XEN full-virtualization (HVM) mode, Shadow Page Table (SPT) is employed to manage the memory. A page fault handler *sh_page_fault* is used to handle the page faults. Mapping a guest OS page to a shadow page with different permissions will be trapped into the handler. Similar to the case in para-virtualization, the system will invoke a new hypercall *do_hvm_write_protect_page* to notify the PM that a new signature page has been created, and PM will record the address of PTE of the signature page. When a mapping being trapped, PM will intercept *sh_page_fault* to check if there is a different permission setting about the PTE of signature pages (writeable for guest OS pages while read-only for shadow pages). If so, PM makes sure the physical pages are marked read-only.

**Table 1: The Configurations of experiment environment.**

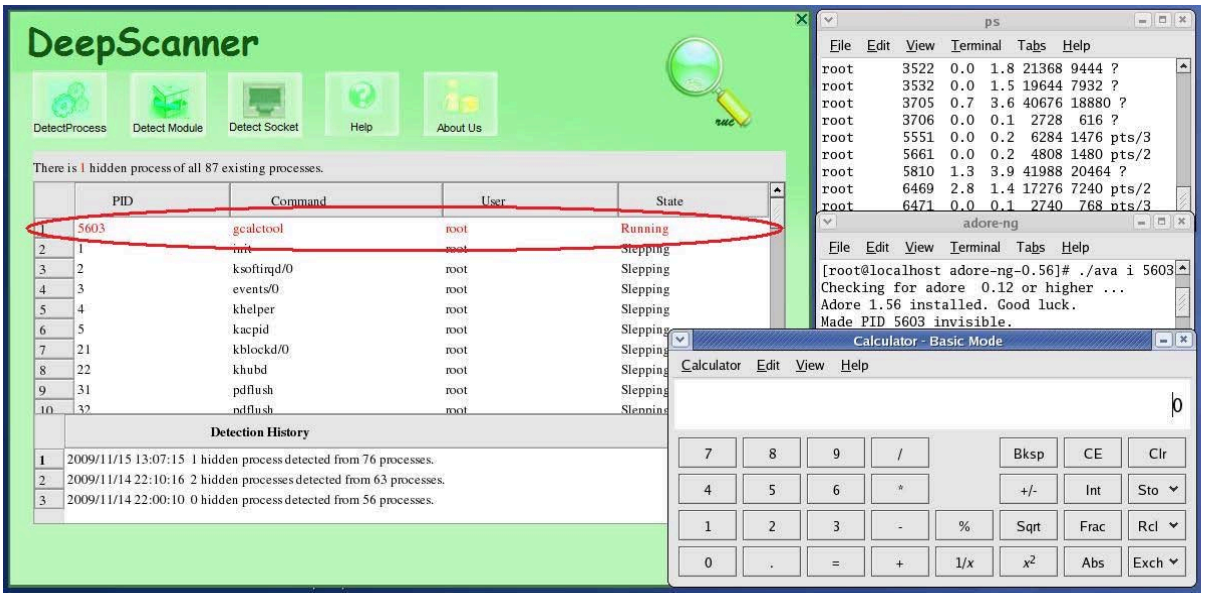
| Hardware Configuration | |
|---|---|
| ***Processor*** | Intel Core(TM2) T5600, 1.83GHz |
| ***RAM*** | 2.0GB |
| ***Storage*** | 60GB IDE |
| **Hypervisor Configuration** | |
| ***Hypervisor*** | Xen |
| ***Version*** | 3.1.0 |
| **Host OS Configuration** | |
| ***OS Version*** | Red Hat Enterprise Linux 5 |
| ***Kernel Version*** | 2.6.18 |
| ***Xen supported*** | Domain 0 |
| **Guest OS Configuration I** | |
| ***OS Version*** | Red Hat Enterprise Linux 4 |
| ***Kernel Version*** | 2.6.9 |
| ***Xen supported*** | Paravirtualization Domain U |
| **Guest OS Configuration II** | |
| ***OS Version*** | Red Hat 7.3 |
| ***Kernel Version*** | 2.4.18 |
| ***Xen supported*** | HVM |

## 5. EVALUATION

In this section, we present our experiments and results. In particular, we have conducted two sets of experiments. The first set of experiments is to evaluate DeepScanner's effectiveness in detecting seven real-world rootkits. These rootkits cover the major stealth techniques (KOH and DKOM) currently employed by malwares. DeepScanner turns out to

Table 2: Effectiveness of DeepScanner in detecting 7 real-world rootkits.

| Rootkit | Kernal | Hidden Function | | | Stealth Techniques | Detection Result | | |
|---|---|---|---|---|---|---|---|---|
| | | *process* | *socket* | *module* | | *process* | *socket* | *module* |
| adore-0.42 | Linux 2.4 | Y | Y | Y | DOH, DKOM | √ | √ | √ |
| adore-ng-0.56 | Linux 2.6 | Y | Y | Y | DOH, DKOM | √ | √ | √ |
| knark-2.4.3 | Linux 2.4 | Y | Y | Y | DOH, DKOM | √ | √ | √ |
| wnps-0.26 | Linux 2.6 | Y | Y | Y | DOH, DKOM | √ | √ | √ |
| enyelkm-1.1.2 | Linux 2.6 | Y | N | Y | DOH | √ | | √ |
| hp-1.0.0 | Linux 2.4 | Y | N | N | DKOM | √ | | |
| modhide-1.0 | Linux 2.4 | N | N | Y | DKOM | | | √ |



There is 1 hidden process of all 87 existing processes.

| | PID | Command | User | State |
|---|---|---|---|---|
| 1 | 5603 | gcalctool | root | Running |
| 2 | 1 | init | root | Slepping |
| 3 | 2 | ksoftirqd/0 | root | Slepping |
| 4 | 3 | events/0 | root | Slepping |
| 5 | 4 | khelper | root | Slepping |
| 6 | 5 | kacpid | root | Slepping |
| 7 | 21 | kblockd/0 | root | Slepping |
| 8 | 22 | khubd | root | Slepping |
| 9 | 31 | pdflush | root | Slepping |
| 10 | 32 | pdflush | root | Slepping |

| | Detection History |
|---|---|
| 1 | 2009/11/15 13:07:15 1 hidden process detected from 76 processes. |
| 2 | 2009/11/14 22:10:16 2 hidden processes detected from 63 processes. |
| 3 | 2009/11/14 22:00:10 0 hidden process detected from 56 processes. |

```
root      3522  0.0  1.8 21368 9444 ?
root      3532  0.0  1.5 19644 7932 ?
root      3705  0.7  3.6 40676 18880 ?
root      3706  0.0  0.1  2728  616 ?
root      5551  0.0  0.2  6284 1476 pts/3
root      5661  0.0  0.2  4808 1480 pts/2
root      5810  1.3  3.9 41988 20464 ?
root      6469  2.8  1.4 17276 7240 pts/2
root      6471  0.0  0.1  2740  768 pts/3
```

```
[root@localhost adore-ng-0.56]# ./ava i 5603
Checking for adore  0.12 or higher ...
Adore 1.56 installed. Good luck.
Made PID 5603 invisible.
```

Figure 12: The Screenshot of the hidden process detection.

be effective in detecting them. We will discuss the detection of the most classical rootkit *adore-ng* in details, and show the detection results of all the rootkits in a table. The second set of experiments is to evaluate DeepScanner's ability against evasion attacks. We develop an experimental rootkit for evasion attacks based on *adore-ng* rootkit. The results show that it could not evade DeepScanner without breaking the normal functions of target objects and the system.

We perform our experiments in two versions of Linux kernel (2.4.18 and 2.6.9) in order to test our approach with more available rootkits. The detail configurations of our experiment environment are listed in Table 1.

## 5.1 Real-world Rootkits Experiments

We have evaluated DeepScanner with 7 real-world Linux rootkits shown in Table 2. These rootkits all have the function of hiding one or more kinds of system objects, including processes, sockets and kernel modules. The major stealth techniques (KOH and DKOM) were covered by these rootkits. DeepScanner successfully detected all stealthy objects hidden by these rootkits via scanning system kernel memory with the four invariants discussed above. *The remarkable thing is that DeepScanner's precision is perfect, it doesn't produce any false negatives or false positives.* In the following, we describe in detail our experiments with a classical rootkit.

The *adore-ng* rootkit infects the kernel as an LKM. It has the function of hiding processes, sockets and kernel modules. The stealth technique it adopts to hide processes is KOH targeting kernel function pointers *proc_root.proc_iops->readdir* and *proc_root_inode_operations->loopup*. The same way it works to hide sockets targeting the kernel function pointer *proc_net->get_info*. In addition, *adore-ng* employs DKOM technique to hide kernel modules, it unlinks the target module from the module list maintained by Linux kernel.

We use DeepScanner to detect the objects hidden by *adore-ng* rootkit. The screenshot of the hidden process detection is shown in Figure 12. We can observe that a calculator process (PID: 5603) is still running even though that *adore-ng* rootkit hides it. It cannot be found in the result of *ps* utility while we can get it from the output of DeepScanner. According to the cross-view technique, scanner could definitely draw a conclusion that the process with PID 5603 is a hidden process. In DeepScanner console, hidden objects will be shown at the top of the objects list and highlighted.

## 5.2 Performance

We measured two aspects of DeepScanner's performance: (a) detection time, i.e., the time taken to scan memory to discover hidden objects; and (b) performance overhead, i.e., the overhead on the target system as a result of signature pages protection.

**Detection Time.** DeepScanner employs a directive scanning strategy. DeepScanner only fetches candidate memory block from specific parts of kernel memory space rather than traversing the whole kernel memory byte by byte, i.e., it only checks the memory blocks in the slab list with specific item size to find the *socket_alloc* data structure, the memory blocks with 8kb-aligned start addresses to find *thread_info* data structure, and identifies imported signature pages page by page. Consequently, the scanning of DeepScanner is very fast, the time of scanning processes, sockets and kernel modules is no more than 1 second in the experiment environment.

**Performance Overhead.** Before enforcing the signature pages protection, we use Unixbench to measure the performance of the whole system. The final score is 94.2. After install signature pages protection mechanism, the final score is reduced to 93.4 (only fell less than 1%). The performance overhead is negligible and acceptable completely.

## 5.3 Experiments for Evasion Attacks

In order to evaluate the DeepScanner's ability against evasion attacks, we implement an experimental rootkit based on *adore-ng*. It makes evasion attacks by modifying the fields related to signatures. This set of experiments consists of 4 parts. Each one attempts to evade the detection via violating one of the four invariants used by DeepScanner. The experiment results show that the rootkit cannot evade DeepScanner without breaking the normal functions of target objects and the system.

1) Violation of Invariant 1.
The experimental rootkit attempts to evade hidden socket detection by breaking the reference chain about the *socket_alloc* data structure. It does this via setting any one link pointer field of the chain with another arbitrary address value to break the original reference, i.e., *file*, *f_dentry*, or *d_inode* pointer is set to an accessible address different from original value. After doing this, the connection becomes unstable and broken finally.

2) Violation of Invariant 2.
The experimental rootkit attempts to evade hidden process detection by breaking the cross-reference relationship between a *task_struct* data structure *t* and a related *thread_info* data structure *th*. It does so by setting either (*t.thread_info*)->*task* or (*th.task*)->*thread_info* to another value. As a result, system crashes immediately.

3) Violation of Invariant 3.
Similar to 1), the experimental rootkit attempts to evade hidden socket detection by breaking the reference relationship of a *socket_alloc* data structure *sa*, a related *task_struct* data structure *t*, and a related file data structure *f*. If it does so by pointing either socket-related (*t.files*)->*fd_array*[*i*] or (*f.f_dentry*)->*d_inode* to another place, the socket will be disconnected. If it does so by modifying the (*sa.socket*)->*file* pointer, the system will crash.

4) Violation of Invariant 4.
The experimental rootkit attempts to evade hidden module detection by tampering the signature pages. Due to all signature pages without write permission in the guest OS level, directly modifying the content in a signature page will incur a segmentation fault. Furthermore, the experimental rootkit also attempts to set the signature pages to writeable by modifying the related PTE. Under the protection of the monitor deployed in XEN hypervisor, any modification towards the protected PTE is terminated and fails.

## 6. LIMITATIONS AND FUTURE WORK

The fundamental limitation of our approach is that the signatures are concluded based on the manual analysis to kernel data structures. Although there is a small number of desired objects need to be analyzed, it is only feasible for an open source OS, e.g., Linux. For a closed source OS, like Windows, we need to develop an automatic technique for signatures generation similar to [12][16]. In the future, we want to introduce dynamic analysis to generate inter-structure signatures for closed source OS. The fuzzing technique will be employed to explore the relationships between multiple data structures.

Because the main purpose of DeepScanner is to demonstrate the effectiveness of inter-structure signatures and imported signatures, its detection mechanism is currently implemented as an LKM. A limitation of DeepScanner is that detection is enforced in target OS kernel. The detection may be compromised by rootkits designed specifically to tamper with the detection mechanisms. Cooperating with virtual machine techniques, we can also develop an out-of-box scanner. It is advisable that security mechanisms are deployed in hypervisors so that they can be shielded from malicious attacks coming from a guest virtual machine, even if the guest operating system kernel is compromised. In the future, we'll port DeepScanner to hypervisor to counter possible attacks.

## 7. CONCLUSIONS

Scanning memory for object signatures to detect stealthy malwares has been proven to be a useful approach. It is critical to construct robust scanning signatures for detection. Traditional approaches choose fields from a single data structure to be scanning signatures. But, our analysis and experiments prove that this way is limited or even impossible as to some critical system object structures, e.g., the *module* data structure in Linux kernel. In this paper, two new concepts are introduced: inter-structure signature and imported signature. Based on the two concepts, the space of signatures is extended to involving relationships of multiple data structures or importing artificial signatures rather than being limited in a single data structure. In this paper, we provide four invariants as signatures to detect hidden processes, sockets, and kernel modules in Linux. Accordingly, we implement a detection prototype system DeepScanner and a hypervisor-based monitor to protect imported signatures. The experiment results show that DeepScanner can effectively and efficiently detect stealthy malwares and it can also resist evasion attacks. Our works provide a new method to construct robust signatures. The invariants presented in this paper can be used immediately by applications to locate processes, sockets and kernel modules in memory.

## 8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. The work has been supported by the National Natural Science Foundation of China under Grant No. 60873213, 91018008, 60703103 and 61070192, the Beijing Science Foundation under Grant No. 4082018 and Shanghai Key Laboratory of Intelligent Information Processing, China. Grant No. IIPL-09-006.

## 9. REFERENCES

[1] Bypassing Klister. http://www.rootkit.com/newsread.php?newsid=235.

[2] Chkrootkit. http://www.chkrootkit.org.

[3] KSTAT. http://www.s0ftpj.org/tools/kstat24_v1.1-2.tgz.

[4] QEMU. http://www.qemu.org.

[5] Rkhunter. http://www.rootkit.nl/projects/rootkit_hunter.html.

[6] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th ACM Conference on Computer Security Applications Conference (ACSAC)*, 2008.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2003.

[8] C. Betz. MemParser. http://sourceforge.net/projects/memparser.

[9] C. Bugcheck. Grepexec: Grepping executive objects from pool memory. http://www.uninformed.org/?v=4&a=2&t=sumry.

[10] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[11] D. Chisnall. *The definitive guide to the xen hypervisor, 2nd edn.* Prentice Hall Press, 2007.

[12] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[14] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[15] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM Conference on Virtual Execution Environments (VEE)*, 2008.

[16] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[17] N. Petroni Jr, T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[18] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. In *Proceedings of the 4th IEEE Conference on Availability, Reliability and Security (ARES)*, 2009.

[19] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[20] J. Rutkowska. Klister v0.3. https://www.rootkit.com/newsread.php?newsid=51.

[21] A. Schuster. Searching for processes and threads in Microsoft Windows memory dumps. *Digital Investigation*, 3:10–16, 2006.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[23] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[24] A. Walters and N. Petroni Jr. Volatools: Integrating volatile memory forensics into the digital investigation process. *Black Hat DC*, 2007.

[25] Y. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with Strider GhostBuster. In *Proceedings of the 35th IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[26] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[27] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[28] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*, 2008.