Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory

Junghwan Rhee[1], Ryan Riley[2], Dongyan Xu[1], and Xuxian Jiang[3]

[1] Purdue University
{rhee,dxu}@cs.purdue.edu
[2] Qatar University
ryan.riley@qu.edu.qa
[3] North Carolina State University
jiang@cs.ncsu.edu

Abstract. Dynamic kernel memory has been a popular target of recent kernel malware due to the difficulty of determining the status of volatile dynamic kernel objects. Some existing approaches use kernel memory mapping to identify dynamic kernel objects and check kernel integrity. The snapshot-based memory maps generated by these approaches are based on the kernel memory which may have been manipulated by kernel malware. In addition, because the snapshot only reflects the memory status at a single time instance, its usage is limited in temporal kernel execution analysis. We introduce a new runtime kernel memory mapping scheme called *allocation-driven mapping*, which systematically identifies dynamic kernel objects, including their types and lifetimes. The scheme works by capturing kernel object allocation and deallocation events. Our system provides a number of unique benefits to kernel malware analysis: (1) an un-tampered view wherein the mapping of kernel data is unaffected by the manipulation of kernel memory and (2) a temporal view of kernel objects to be used in temporal analysis of kernel execution. We demonstrate the effectiveness of allocation-driven mapping in two usage scenarios. First, we build a hidden kernel object detector that uses an un-tampered view to detect the data hiding attacks of 10 kernel rootkits that directly manipulate kernel objects (DKOM). Second, we develop a temporal malware behavior monitor that tracks and visualizes malware behavior triggered by the manipulation of dynamic kernel objects. Allocation-driven mapping enables a reliable analysis of such behavior by guiding the inspection only to the events relevant to the attack.

Keywords: Kernel memory mapping, kernel malware analysis, virtualization.

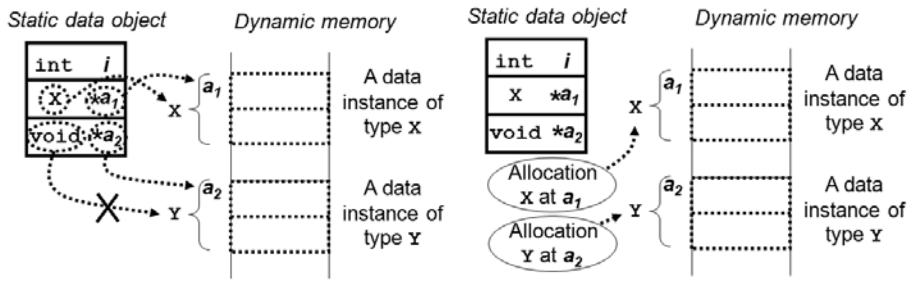
1 Introduction

Dynamic kernel memory is where the majority of kernel data resides. Operating system (OS) kernels frequently allocate and deallocate numerous dynamic objects of various types. Due to the complexity of identifying such objects at runtime, dynamic kernel memory is a source of many kernel security and reliability problems. For instance, an increasing amount of kernel malware targets dynamic kernel objects [4,10,18,23]; and many kernel bugs are caused by dynamic memory errors [13,27,28].

Advanced kernel malware uses stealthy techniques such as directly manipulating kernel data (i.e., DKOM [4]) or overwriting function pointers (i.e., KOH [10]) located in dynamic kernel memory. This allows attacks such as process hiding and kernel-level control flow hijacking. These anomalous kernel behaviors are difficult to analyze because they involve manipulating kernel objects that are dynamically allocated and deallocated at runtime; unlike persistent kernel code or static kernel data that are easier to locate, monitor, and protect.

To detect these attacks, some existing approaches use kernel memory mapping based on the contents of runtime memory snapshots [1,5,16] or memory access traces [23,31]. These approaches commonly identify a kernel object by projecting the type and address of a pointer onto the memory. However, such a technique may not always be accurate – for example, when an object is type cast to a generic type or when an embedded list structure is used as part of larger data types. In benign kernel execution, such inaccuracy can be corrected [5]; but it becomes a problem in malware analysis as the memory contents may have been manipulated by kernel malware. For example, a DKOM attack to hide a process may modify the `next_task` and `prev_task` pointers in the process list. This causes the process to disappear from the OS view as well as from the kernel memory map. To detect this attack, some existing approaches rely on data invariants such as that the list used for process scheduling should match the process list. However, not every data structure has an invariant. Additionally, the kernel memory map generated from a snapshot [1,5,16] reflects kernel memory status at a specific time instance. Therefore, the map is of limited usage in analyzing the kernel execution. Some mapping approaches are based on logging malware memory accesses [23,31] and thus provide temporal information. However they only cover objects accessed by the malware code and cannot properly handle certain attack patterns due to assumptions in its mapping algorithm [21].

In this paper, we present a new kernel memory mapping scheme called *allocation-driven mapping* that complements the existing approaches. Our scheme identifies dynamic kernel objects by capturing their allocations and does not rely on the runtime content of kernel memory to construct the kernel object map. As such, the map is resistant to attacks that manipulate the kernel memory. On top of our scheme, we build a hidden kernel object detector that uses the un-tampered view of kernel memory to detect DKOM data hiding attacks without requiring kernel object-specific invariants. In addition, our scheme keeps track of each kernel object's life time. This temporal property is useful in the analysis of kernel/kernel malware execution. We also build a temporal malware behavior monitor that systematically analyzes the impact of kernel malware attacks via dynamic kernel memory using a kernel execution trace. We address a challenge in the use of kernel memory mapping for temporal analysis of kernel execution: A dynamic memory address may correspond to different kernel objects at different times because of the runtime allocation and deallocation events. This problem can be handled by allocation-driven mapping. The lifetime of a dynamic kernel object naturally narrows the scope of a kernel malware analysis.



(a) Type-projection mapping (b) Allocation-driven mapping

Fig. 1. Illustration of kernel memory mapping approaches. a_1 and a_2 represent kernel memory addresses. X and Y are data types for kernel objects.

The contributions of this paper are summarized as follows:

- We present a new kernel memory mapping scheme called allocation-driven mapping that has the following properties desirable for kernel malware analysis: un-tampered identification of kernel objects and temporal status of kernel objects.
- We implement allocation-driven mapping at the virtual machine monitor (VMM) level. The identification and tracking of kernel objects take place in the VMM without modification to the guest OS.
- We develop a hidden kernel object detector that can detect DKOM data hiding attacks without requiring data invariants. The detector works by comparing the status of the un-tampered kernel map with that of kernel memory.
- We develop a malware behavior monitor that uses a temporal view of kernel objects in the analysis of kernel execution traces. The lifetimes of dynamic kernel objects in the view guide the analysis to the events triggered by the objects manipulated by the malware.

We have implemented a prototype of allocation-driven mapping called LiveDM (Live Dynamic kernel memory Map). It supports three off-the-shelf Linux distributions. LiveDM is designed for use in non-production scenarios such as honeypot monitoring, kernel malware profiling, and kernel debugging.

2 Background – Kernel Memory Mapping

There have been several approaches [1,5,16,23,31] that leverage kernel memory mapping to test the integrity of OS kernels and thereby detect kernel malware. These approaches (similar to garbage collection [3,19]) commonly identify kernel objects by recursively traversing pointers in the kernel memory starting from static objects. A kernel object is identified by projecting the address and type of a traversed pointer onto memory; thus, we call this mechanism *type-projection mapping*. For example, in Fig. 1(a) the mapping process starts by evaluating the pointer fields of the static data object. When the second field of this object is traversed, the type X of the pointer is projected onto the memory located in the obtained address a_1, identifying an instance of type X.

The underlying hypothesis of this mapping is that the traversed pointer's type accurately reflects the type of the projected object. In practice there are several cases when this may not be true. First, if an object allocated using a specific type is later cast to a generic type, then this mapping scheme cannot properly identify the object using that pointer. For instance, in Fig. 1(a) the third field of the static object cannot be used to identify the `Y` instance due to its generic `void*` type. Second, in modern OSes many kernel objects are linked using embedded list structures which connect the objects using list types. When these pointers are traversed, the connected objects are inaccurately identified as list objects. KOP [5] addresses these problems by generating an extended type graph using static analysis. Some other approaches rely on manual annotations.

When type-projection mapping is used against kernel malware, these problems may pose concerns as such inaccuracy can be deliberately introduced by kernel malware. In type-projection mapping, *the kernel memory map is based on the content of the kernel memory*, which may have been manipulated by the kernel malware. This property may affect the detection of kernel rootkits that hide kernel objects by directly manipulating pointers. To detect such attacks, a detector needs to rely on not only the kernel memory map but also additional knowledge that reveals the anomalous status of the hidden objects. For this purpose, several approaches [1,5,18] use data structure invariants. For example, KOP [5] detects a process hidden by the FU Rootkit [4] by using the invariant that there are two linked lists regarding process information which are supposed to match, and one of them is not manipulated by the attack. However, a data invariant is specific to semantic usage of a data structure and may not be applicable to other data structures. For type-projection mapping, it is challenging to detect data hiding attacks that manipulate a simple list structure (such as the kernel module list in Linux) without an accompanying invariant.

In general, we can categorize these approaches into two categories based on whether they make use of a static snapshot or dynamic runtime memory access trace.

2.1 Static Type-Projection Mapping

This approach uses a memory snapshot to generate a kernel memory map. SBCFI [16] constructs a map to systematically detect the violation of persistent control flow integrity. Gibraltar [1] extracts data invariants from kernel memory maps to detect kernel rootkits. A significant advantage of this approach is the low cost to generate a memory snapshot. A memory snapshot can be generated using an external monitor such as a PCI interface [1], a memory dump utility [5], or a VMM [16], and the map is generated from the snapshot later.

The memory snapshot is generated at a specific time instance (asynchronously); therefore, its usage is limited for analyzing kernel execution traces where dynamic kernel memory status varies over time. The same memory address, for example, could store different dynamic kernel objects over a period of time (through a series of deallocations and reallocations). The map cannot be used to properly determine what data was stored at that address at a specific time. We call this a *dynamic data identity problem*, and it occurs when an asynchronous kernel memory map is used for inspection of dynamic memory status along the kernel execution traces.

2.2 Dynamic Type-Projection Mapping

This mapping approach also uses the type-projection mechanism to identify kernel objects, but its input is the trace of memory accesses recorded over runtime execution instead of a snapshot. By tracking the memory accesses of malware code, this approach can identify the list of kernel objects manipulated by the malware. PoKeR [23] and Rkprofiler [31] use this approach to profile dynamic attack behavior of kernel rootkits in Linux and Windows respectively.

Since a runtime trace is used for input, this approach can overcome the asynchronous nature of static type-projection mapping. Unfortunately, current work only focuses on the data structures accessed by malware code, and may not capture other events. For example, many malware programs call kernel functions during the attack or exploit various kernel bugs, and these behaviors may appear to be part of legitimate kernel execution. In these cases, this dynamic type-projection techniques need to track all memory accesses to accurately identify the kernel objects accessed by legitimate kernel execution. Since this process is costly (though certainly possible), it is not straightforward for this approach to expand the coverage of the mapped data to all kernel objects.

3 Design of LiveDM

In this section, we first introduce the allocation-driven mapping scheme, based on which our LiveDM system is implemented. We then present key enabling techniques to implement LiveDM.

3.1 Allocation-Driven Mapping Scheme

Allocation-driven mapping is a kernel memory mapping scheme that generates a kernel object map by *capturing the kernel object allocation and deallocation events* of the monitored OS kernel. LiveDM uses a VMM in order to track the execution of the running kernel. Whenever a kernel object is allocated or deallocated, LiveDM will intercede and capture its address range and the information to derive the data type of the object subject to the event (details in Section 3.2) in order to update the kernel object map. We first present the benefits of allocation-driven mapping over existing approaches. After that we will present the techniques used to implement this mapping scheme.

First, this approach does not rely on any content of the kernel memory which can potentially be manipulated by kernel malware. Therefore, the kernel object map provides *an un-tampered view* of kernel memory wherein the identification of kernel data is not affected by the manipulation of memory contents by kernel malware. This tamper-resistant property is especially effective to detect sophisticated kernel attacks that directly manipulate kernel memory to hide kernel objects. For instance, in the type-projection mapping example (Fig. 1(a)) if the second pointer field of the static object is nullified, the X object cannot be identified because this object cannot be reached by recursively scanning all pointers in the memory. In practice, there can be multiple pointer references to a dynamic object. However, malware can completely isolate an object to be hidden by tampering with all pointers pointing to the object. The address of the hidden object can be safely stored in a non-pointer storage (e.g., `int` or `char`) to avoid being discovered by the type-projection mapping algorithm while it can be used to recover the object when necessary. Many malicious programs carefully control their activities to avoid detection and prolong their stealthy operations, and it is a viable option to suspend a data object in this way temporarily and activate it again when needed [30].

In the allocation-driven mapping approach, however, this attack will not be effective. As shown in Fig. 1(b), each dynamic object is recognized upon its allocation. Therefore the identification of dynamic objects is reliably obtained and protected against the manipulation of memory contents. The key observation is that allocation-driven mapping captures the *liveness status* of the allocated dynamic kernel objects. For malware writers, this property makes it significantly more difficult to manipulate this view. In Section 6.1, we show how this mapping can be used to automatically detect DKOM data hiding attacks without using any data invariant specific to a kernel data structure.

Second, LiveDM reflects a *temporal* status of dynamic kernel objects since it captures their allocation and deallocation events. This property enables the use of the kernel object map in temporal malware analysis where temporal information, such as kernel control flow and dynamically changing data status, can be inspected to understand complicated kernel malware behavior. In Section 2.1, we pointed out that a *dynamic data identity problem* can occur when a snapshot-based kernel memory map is used for dynamic analysis. Allocation-driven mapping provides a solution to this problem by accurately tracking all allocation and deallocation events. This means that even if an object is deallocated and its memory reused for a different object, LiveDM will be able to properly track it.

Third, allocation-driven mapping does not suffer from the casting problem that occurs when an object is cast to a generic pointer because it does not evaluate pointers to construct the kernel object map. For instance, in Fig. 1(b) the `void` pointer in the third field of the static data object does not hinder the identification of the `Y` instance because this object is determined by capturing its allocation. However, we note that another kind of casting can pose a problem: If an object is allocated using a generic type and it is cast to a specific type later, allocation-driven mapping will detect the earlier generic type. However, our study in Section 5 shows that this behavior is unusual in Linux kernels.

There are a number of challenges in implementing the LiveDM system based on allocation-driven mapping. For example, kernel memory allocation functions do not provide a simple way to determine the type of the object being allocated.[1] One solution is to use static analysis to rewrite the kernel code to deliver the allocation types to the VMM, but this would require the construction of a new type-enabled kernel, which is not readily applicable to off-the-shelf systems. Instead, we use a technique that derives data types by using runtime context (i.e., call stack information). Specifically, this technique systematically captures code positions for memory allocation calls by using virtual machine techniques (Section 3.2) and translates them into data types so that OS kernels can be transparently supported without any change in the source code.

[1] Kernel level memory allocation functions are similar to user level ones. The function `kmalloc`, for example, does not take a type but a size to allocate memory.

3.2 Techniques

We employ a number of techniques to implement allocation-driven mapping. At the conceptual level, LiveDM works as follows. First, a set of kernel functions (such as `kmalloc`) are designated as kernel memory allocation functions. If one of these functions is called, we say that an allocation event has occurred. Next, whenever this event occurs at runtime, the VMM intercedes and captures the allocated memory address range and the code location calling the memory allocation function. This code location is referred to as a *call site* and we use it as a unique identifier for the allocated object's type at runtime. Finally, the source code around each call site is analyzed offline to determine the type of the kernel object being allocated.

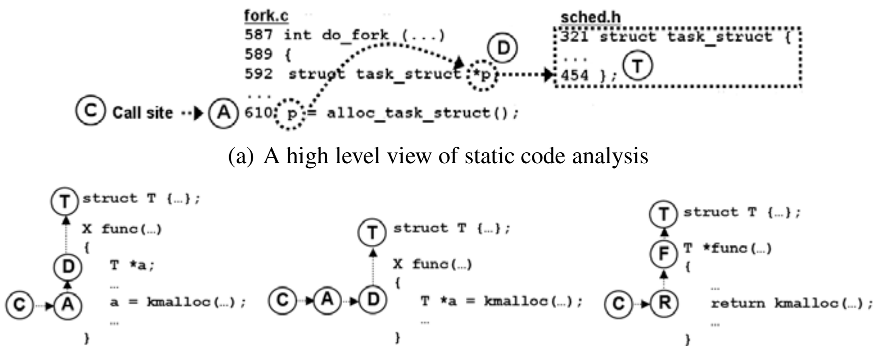
Runtime kernel object map generation. At runtime, LiveDM captures all allocation and deallocation events by interceding whenever one of the allocation/deallocation functions is called. There are three things that need to be determined at runtime: (1) the call site, (2) the address of the objected allocated or deallocated, and (3) the size of the allocated object.

To determine the call site, LiveDM uses the return address of the call to the allocation function. In the instruction stream, the return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map so that the type can be determined during offline source code analysis.

The address and size of objects being allocated or deallocated can be derived from the arguments and return value. For an allocation function, the size is typically given as a function argument and the memory address as the return value. For a deallocation function, the address is typically given as a function argument. These values can be determined by the VMM by leveraging *function call conventions*.[2] Function arguments are delivered through the stack or registers, and LiveDM captures them by inspecting these locations at the entry of memory allocation/deallocation calls. To capture the return value, we need to determine where the return value is stored and when it is stored there. Integers up to 32-bits as well as 32-bit pointers are delivered via the `EAX` register and all values that we would like to capture are either of those types. The return value is available in this register when the allocation function returns to the caller. In order to capture the return values at the correct time the VMM uses a virtual stack. When a memory allocation function is called, the return address is extracted and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercedes and captures the return value from the `EAX` register.

Offline automatic data type determination. The object type information related to kernel memory allocation events is determined using static analysis of the kernel source code offline. Fig. 2(a) illustrates a high level view of our method. First, the allocation call site (C) of a dynamic object is mapped to the source code `fork.c:610` using debugging information found in the kernel binary. This code assigns the address of the allocated memory to a pointer variable at the left-hand side (LHS) of the assignment statement (A). Since this variable's type can represent the type of the allocated memory,

[2] A function call convention is a scheme to pass function arguments and a return value. We use the conventions for the x86 architecture and the `gcc` compiler [8].



```
587 int do_fork (...)
589 {
592 struct task_struct *p
...
610 p = alloc_task_struct();
```

```
321 struct task_struct {
...
454 };
```

```
struct T {...};
X func(...)
{
T *a;
...
a = kmalloc(...);
...
}
```

```
struct T {...};
X func(...)
{
T *a = kmalloc(...);
...
}
```

```
struct T {...};
T *func(...)
{
...
return kmalloc(...);
...
}
```

(a) A high level view of static code analysis

(b) Case 1 (c) Case 2 (d) Case 3

Fig. 2. Static code analysis. C: a call site, A: an assignment, D: a variable declaration, T: a type definition, R: a return, and F: a function declaration

it is derived by traversing the declaration of this pointer (D) and the definition of its type (T). Specifically, during the compilation of kernel source code, a parser sets the dependencies among the internal representations (IRs) of such code elements. Therefore, the type can be found by following the dependencies of the generated IRs.

For type resolution, we enumerate several patterns in the allocation code as shown in Fig. 2(b), 2(c), and 2(d). Case 1 is the typical pattern (C→A→D→T) as just explained. In Case 2, the definition (D) and allocation (A) occur in the same line. The handling of this case is very similar to that of Case 1. Case 3, however, is unlike the first two cases. The pattern in Case 3 does not use a variable to handle the allocated memory address, rather it directly returns the value generated from the allocation call. When a call site (C) is converted to a return statement (R), we determine the type of the allocated memory using the type of the returning function (F). In Fig. 2(d), this pattern is presented as C→R→F→T.

Prior to static code analysis, we generate the set of information about these code elements to be traversed (i.e., C, A, D, R, F, and T) by compiling the kernel source code with the compiler that we instrumented (Section 4).

4 Implementation

Allocation-driven mapping is general enough to work with an OS that follows the standard function call conventions (e.g., Linux, Windows, etc.). Our prototype, LiveDM, supports three off-the-shelf Linux OSes of different kernel versions: Fedora Core 6 (Linux 2.6.18), Debian Sarge (Linux 2.6.8), and Redhat 8 (Linux 2.4.18).

LiveDM can be implemented on any software virtualization system, such as VMware (Workstation and Player) [29], VirtualBox [26], and Parallels [14]. We choose the QEMU [2] with KQEMU optimizer for implementation convenience.

In the kernel source code, many wrappers are used for kernel memory management, some of which are defined as macros or inline functions and others as regular functions.

Macros and inline functions are resolved as the core memory function calls at compile time by a preprocessor; thus, their call sites are captured in the same way as core functions. However, in the case of regular wrapper functions, the call sites will belong to the wrapper code.

To solve this problem, we take two approaches. If a wrapper is used only a few times, we consider that the type from the wrapper can indirectly imply the type used in the wrapper's caller due to its limited use. If a wrapper is widely used in many places (e.g., `kmem_cache_alloc` – a slab allocator), we treat it as a memory allocation function. Commodity OSes, which have mature code quality, have a well defined set of memory wrapper functions that the kernel and driver code commonly use. In our experience, capturing such wrappers, in addition to the core memory functions, can cover the majority of the memory allocation and deallocation operations.

We categorize the captured functions into four classes: (1) page allocation/free functions, (2) `kmalloc/kfree` functions, (3) `kmem_cache_alloc/free` functions (slab allocators), and (4) `vmalloc/vfree` functions (contiguous memory allocators). These sets include the well defined wrapper functions as well as the core memory functions. In our prototype, we capture about 20 functions in each guest kernel. The memory functions of an OS kernel can be determined from its design specification (e.g., the Linux Kernel API) or kernel source code.

Automatic translation of a call site to a data type requires a kernel binary that is compiled with a debugging flag (e.g., `-g` to `gcc`) and whose symbols are not stripped. Modern OSes, such as Ubuntu, Fedora, and Windows, generate kernel binaries of this form. Upon distribution, typically the stripped kernel binaries are shipped; however, unstripped binaries (or symbol information in Windows) are optionally provided for kernel debugging purposes. The experimented kernels of Debian Sarge and Redhat 8 are not compiled with this debugging flag. Therefore, we compiled the distributed source code and generated the debug-enabled kernels. These kernels share the same source code with the distributed kernels, but the offset of the compiled binary code can be slightly different due to the additional debugging information.

For static analysis we use a `gcc` [8] compiler (version 3.2.3) that we instrumented to generate IRs for the source code of the experimented kernels. We place hooks in the parser to extract the abstract syntax trees for the code elements necessary in the static code analysis.

5 Evaluation

In this section, we evaluate the basic functionality of LiveDM with respect to the identification of kernel objects, casting code patterns, and the performance of allocation-driven mapping. The guest systems are configured with 256MB RAM and the host machine has a 3.2Ghz Pentium D CPU and 2GB of RAM.

Identifying dynamic kernel objects. To demonstrate the ability of LiveDM to inspect the runtime status of an OS kernel, we present a list of important kernel data structures captured during the execution of Debian Sarge OS in Table 1. These data structures manage the key OS status such as process information, memory mapping of each process, and the status of file systems and network which are often targeted by kernel malware

Table 1. A list of core dynamic kernel objects and the source code elements used to derive their data types in static analysis. (OS: Debian Sarge).

	Call Site	Declaration	Data Type	Case	#Objects
Task/Sig	kernel/fork.c:248	kernel/fork.c:243	task_struct	1	66
	kernel/fork.c:801	kernel/fork.c:795	sighand_struct	1	63
	fs/exec.c:601	fs/exec.c:587	sighand_struct	1	1
	kernel/fork.c:819	kernel/fork.c:813	signal_struct	1	66
Memory	arch/i386/mm/pgtable.c:229	arch/i386/mm/pgtable.c:229	pgd_t	2	54
	kernel/fork.c:433	kernel/fork.c:431	mm_struct	1	47
	kernel/fork.c:559	kernel/fork.c:526	mm_struct	1	7
	kernel/fork.c:314	kernel/fork.c:271	vm_area_struct	1	149
	mm/mmap.c:923	mm/mmap.c:748	vm_area_struct	1	1004
	mm/mmap.c:1526	mm/mmap.c:1521	vm_area_struct	1	5
	mm/mmap.c:1722	mm/mmap.c:1657	vm_area_struct	1	48
	fs/exec.c:402	fs/exec.c:342	vm_area_struct	1	47
File system	kernel/fork.c:677	kernel/fork.c:654	files_struct	1	54
	kernel/fork.c:597	kernel/fork.c:597	fs_struct	2	53
	fs/file_table.c:76	fs/file_table.c:69	file	1	531
	fs/buffer.c:3062	fs/buffer.c:3062	buffer_head	2	828
	fs/block_dev.c:232	fs/block_dev.c:232	bdev_inode	2	5
	fs/dcache.c:692	fs/dcache.c:689	dentry	1	4203
	fs/inode.c:112	fs/inode.c:107	inode	1	1209
	fs/namespace.c:55	fs/namespace.c:55	vfsmount	2	16
	fs/proc/inode.c:93	fs/proc/inode.c:90	proc_inode	1	237
	drivers/block/ll_rw_blk.c:1405	drivers/block/ll_rw_blk.c:1405	request_queue_t	2	18
	drivers/block/ll_rw_blk.c:2950	drivers/block/ll_rw_blk.c:2945	io_context	1	10
Network	net/socket.c:279	net/socket.c:278	socket_alloc	1	12
	net/core/sock.c:617	net/core/sock.c:613	sock	1	3
	net/core/dst.c:125	net/core.dst.c:119	dst_entry	1	5
	net/core/neighbour.c:265	net/core/neighbour.c:254	neighbour	1	1
	net/ipv4/tcp_ipv4.c:134	net/ipv4/tcp_ipv4.c:133	tcp_bind_bucket	2	4
	net/ipv4/fib_hash.c:586	net/ipv4/fib_hash.c:461	fib_node	1	9

and kernel bugs [13,15,16,17,18,23,27,28]. Kernel objects are recognized using allocation call sites shown in column **Call Site** during runtime. Using static analysis, this information is translated into the data types shown in column **Data Type** by traversing the allocation code and the declaration of a pointer variable or a function shown in column **Declaration**. Column **Case** shows the kind of the allocation code pattern described in Section 3.2. The number of the identified objects for each type in the inspected runtime status is presented in column **#Objects**. At that time instance, LiveDM identified total of 29488 dynamic kernel objects with their data types derived from 231 allocation code positions.

In order to evaluate the accuracy of the identified kernel objects, we build a reference kernel where we modify kernel memory functions to generate a log of dynamic kernel objects and run this kernel in LiveDM. We observe that the dynamic objects from the log accurately match the live dynamic kernel objects captured by LiveDM. To check the type derivation accuracy, we manually translate the captured call sites to data types by traversing kernel source code as done by related approaches [5,7]. The derived types at the allocation code match the results from our automatic static code analysis.

Code patterns casting objects from generic types to specific types. In Section 3.1, we discussed that allocation-driven mapping has no problem handling the situation where a specific type is cast to a generic type, but casting from generic types to specific types can be a problem. In order to estimate how often this type of casting occurs, we manually checked all allocation code positions where the types of kernel objects are derived for the inspected status. We checked for the code pattern that memory is allocated using a generic pointer and then the address is cast to the pointer of a more specific type. Note that this pattern does not include the use of generic pointers for generic purposes. For example, the use of void or integer pointers for bit fields or buffers is a valid use of generic pointers. Another valid use is kernel memory functions that internally handle pre-typed memory using generic pointers to retail it to various types. We found 25 objects from 10 allocation code positions (e.g., `tty_register_driver` and `vc_allocate`) exhibiting this behavior at runtime. Such objects are not part of the core data structures shown in Table 1, and they account for only 0.085% of all objects. Hence we consider them as non-significant corner cases. Since the code positions where this casting occurs are available to LiveDM, we believe that the identification of this behavior and the derivation of a specific type can be automated by performing static analysis on the code after the allocation code.

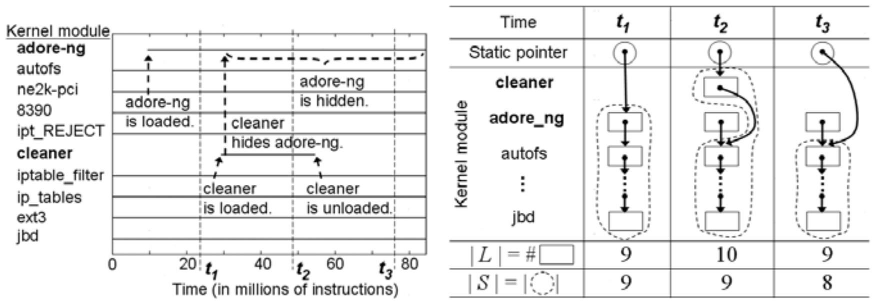
Performance of allocation-driven mapping. Since LiveDM is mainly targeted for non-production environments such as honeypots and kernel debugging systems, performance is not a primary concern. Still, we would like to provide a general idea of the cost of allocation-driven mapping. In order to measure the overhead to generate a kernel object map at runtime, we ran three benchmarks: compiling the kernel source code, UnixBench (Byte Magazine Unix Benchmark 5.1.2), and nbench (BYTEmark* Native Mode Benchmark version 2). Compared to unmodified QEMU, our prototype incurs (in the worst case) 41.77% overhead for Redhat 8 (Linux 2.4) and 125.47% overhead for Debian Sarge (Linux 2.6). For CPU intensive workload such as nbench, the overhead is near zero because the VMM rarely intervenes. However, applications that use kernel services requiring dynamic kernel memory have higher overhead. As a specific example, compiling the Linux kernel exhibited an overhead of 29% for Redhat 8 and 115.69% for Debian Sarge. It is important to note that these numbers measure overhead when compared to an unmodified VMM. Software based virtualization will add additional overhead as well. For the purpose of inspecting fine-grained kernel behavior in non-production environments, we consider this overhead acceptable. The effects of overhead can even be minimized in a production environment by using decoupled analysis [6].

6 Case Studies

We present two kernel malware analysis systems built on top of LiveDM: a hidden kernel object detector and a temporal malware behavior monitor. These systems highlight the new properties of allocation-driven mapping which are effective for detection and analysis of kernel malware attacks.

6.1 Hidden Kernel Object Detector

One problem with static type-projection approaches is that they are not able to detect dynamic kernel object manipulation without some sort of data invariant. In this section



(a) Temporal live status of kernel modules based on allocation-driven mapping. (b) Live set (L) and scanned set (S) for kernel modules at t_1, t_2, and t_3.

Fig. 3. Illustration of the kernel module hiding attack by `cleaner` rootkit. Note that the choice of t_1, t_2, and t_3 is for the convenience of showing data status and irrelevant to the detection. This attack is detected based on the difference between L and S.

we present a hidden kernel object detector built on top of LiveDM that does not suffer from this limitation.

Leveraging the un-tampered view. Some advanced DKOM-based kernel rootkits hide kernel objects by simply removing all references to them from the kernel's dynamic memory. We model the behavior of this type of DKOM data hiding attack as a data anomaly in a list. If a dynamic kernel object does not appear in a kernel object list, then it is orphaned and hence an anomaly. As described in Section 3.1, allocation-driven mapping provides an un-tampered view of the kernel objects not affected by manipulation of the actual kernel memory content. Therefore, if a kernel object appears in the LiveDM-generated kernel object map but cannot be found by traversing the kernel memory, then that object has been hidden. More formally, for a set of dynamic kernel objects of a given data type, a live set L is the set of objects found in the kernel object map. A scanned set S is the set of kernel objects found by traversing the kernel memory as in the related approaches [1,5,16]. If L and S do not match, then a data anomaly will be reported.

This process is illustrated in the example of `cleaner` rootkit that hides the `adore-ng` rootkit module (Fig. 3). Fig. 3(a) presents the timeline of this attack using the lifetime of kernel modules. Fig. 3(b) illustrates the detailed status of kernel modules and corresponding L and S at three key moments. Kernel modules are organized as a linked list starting from a static pointer variable. When the `cleaner` module is loaded after the `adore-ng` module, it modifies the linked list to bypass the `adore-ng` module entry (shown at t_2). Therefore, when the `cleaner` module is unloaded, the `adore-ng` module disappears from the module list (t_3). At this point in time the scanned set S based on static type-projection mapping has lost the hidden module, but the live set L keeps the view of all kernel modules alive. Therefore, the monitor can detect a hidden kernel module due to the condition, $|L| \neq |S|$.

Detecting DKOM data hiding attacks. There are two dynamic kernel data lists which are favored by rootkits as attack targets: the kernel module list and the process control

Table 2. DKOM data hiding rootkit attacks that are automatically detected by comparing LiveDM-generated view (L) and kernel memory view (S)

Rootkit Name	$\|L\|$ - $\|S\|$	Manipulated Data		Operating System	Attack Vector
		Type	Field		
`hide_lkm`	# of hidden modules	`module`	`next`	Redhat 8	`/dev/kmem`
`fuuld`	# of hidden PCBs	`task_struct`	`next_task`, `prev_task`	Redhat 8	`/dev/kmem`
`cleaner`	# of hidden modules	`module`	`next`	Redhat 8	LKM
`modhide`	# of hidden modules	`module`	`next`	Redhat 8	LKM
`hp 1.0.0`	# of hidden PCBs	`task_struct`	`next_task`, `prev_task`	Redhat 8	LKM
`linuxfu`	# of hidden PCBs	`task_struct`	`next_task`, `prev_task`	Redhat 8	LKM
`modhide1`	1 (rootkit self-hiding)	`module`	`next`	Redhat 8	LKM
`kis 0.9` (server)	1 (rootkit self-hiding)	`module`	`next`	Redhat 8	LKM
`adore-ng-2.6`	1 (rootkit self-hiding)	`module`	`list.next`, `list.prev`	Debian Sarge	LKM
`ENYELKM 1.1`	1 (rootkit self-hiding)	`module`	`list.next`, `list.prev`	Debian Sarge	LKM

block (PCB) list.[3] However other linked list-based data structures can be similarly supported as well. The basic procedure is to generate the live set L and periodically generate and compare with the scanned set S. We tested 8 real-world rootkits and 2 of our own rootkits (`linuxfu` and `fuuld`) previously used in [12,21,23], and these rootkits commonly hide kernel objects by directly manipulating the pointers of such objects. LiveDM successfully detected all these attacks just based on the data anomaly from kernel memory maps and the results are shown in Table 2.

In the experiments, we focus on a specific attack mechanism – data hiding via DKOM – rather than the attack vectors – how to overwrite kernel memory – or other attack features of rootkits for the following reason. There are various attack vectors including the ones that existing approaches cannot handle and they can be easily utilized. Specifically, we acknowledge that the rootkits based on loadable kernel module (LKM) can be detected by code integrity approaches [22,24] with the white listing scheme of kernel modules. However, there exist alternate attack vectors such as `/dev/mem`, `/dev/kmem` devices, return-oriented techniques [11,25], and unproven code in third-party kernel drivers which can elude existing kernel rootkit detection and prevention approaches. We present the DKOM data hiding cases of LKM-based rootkits as part of our results because these rootkits can be easily converted to make use of these alternate attack vectors.

We also include results for two other rootkits that make use of these advanced attack techniques. `hide_lkm` and `fuuld` in Table 2 respectively hide kernel modules and processes without any kernel code integrity violation (via `/dev/kmem`) purely based on DKOM, and current rootkit defense approaches cannot properly detect these attacks. However, our monitor effectively detects all DKOM data hiding attacks regardless of attack vectors by leveraging LiveDM-generated kernel object map. Allocation-driven mapping can uncover the hidden object even in more adversary scenarios. For example, if a simple linked list having no data invariant is directly manipulated without violating kernel code integrity, LiveDM will still be able to detect such an attack and uncover the specific hidden object.

In the experiments that detect rootkit attacks, we generate and compare L and S sets every 10 seconds. When a data anomaly occurs, the check is repeated in 1 second. (The

[3] A process control block (PCB) is a kernel data structure containing administrative information for a particular process. Its data type in Linux is `task_struct`.

repeated check ensures that a kernel data structure was not simply in an inconsistent state during the first scan.) If the anomaly persists, then we consider it as a true positive. With this monitoring policy, we successfully detected all tested DKOM hiding attacks without any false positives or false negatives.

We note that while this section focuses on data hiding attacks based on DKOM, data hiding attacks without manipulating data (such as rootkit code that filters system call results) may also be detected using the LiveDM system. Instead of comparing the un-tampered LiveDM-generated view with the scanned view of kernel memory, one could simply compare the un-tampered view with the user-level view of the system.

6.2 Temporal Malware Behavior Monitor

Kernel rootkit analysis approaches based on dynamic type-projection are able to perform temporal analysis of a running rootkit. One problem with these approaches, however, is that they are only able to track malware actions that occur from injected rootkit code. If a rootkit modifies memory indirectly through other means such as legitimate kernel functions or kernel bugs, these approaches are unable to follow the attack. Allocation-driven mapping does not share this weakness. To further illustrate the strength of allocation-driven mapping, we built a temporal malware behavior monitor (called a temporal monitor or a monitor below for brevity) that uses a kernel object map in temporal analysis of a kernel execution trace.

In this section, we highlight two features that allocation-driven mapping newly provides. First, allocation-driven mapping enables the *use of a kernel object map covering all kernel objects in temporal analysis*; therefore for any given dynamic kernel object we can inspect how it is being used in the dynamic kernel execution trace regardless of the accessing code (either legitimate or malicious), which is difficult for both static and dynamic type-projection approaches. Second, the data lifetime in allocation-driven mapping lets the monitor *avoid the dynamic data identity problem* (Section 2.1) which can be faced by an asynchronous memory map.

Systematic visualization of malware influence via dynamic kernel memory. Our monitor systematically inspects and visualizes the influence of kernel malware attacks targeting dynamic kernel memory. To analyze this dynamic attack behavior, we generate a full system trace including the kernel object map status, the executed code, and the memory accesses during the experiments of kernel rootkits. When a kernel rootkit attack is launched, if it violates kernel code integrity, the rootkit code is identified by using our previous work, NICKLE [22]. Then the temporal monitor systematically identifies all targets of rootkit memory writes by searching the kernel object map. If the attack does not violate code integrity, the proposed technique in the previous section or any other approach can be used to detect the dynamic object under attack. The identified objects then become the *causes* of malware behavior and their *effects* are systematically visualized by searching the original and the modified kernel control flow triggered by such objects. For each object targeted by the rootkit, there are typically multiple behaviors using its value. Among those, this monitor samples a pair of behaviors caused by the same code,

Table 3. The list of kernel objects manipulated by `adore-ng` rootkit. (OS: Redhat 8).

Runtime Identification		Offline Data Type Interpretation	
Call Site	**Offset**	**Type / Object (Static, Module object)**	**Field**
`fork.c:610`	`0x4,12c,130`	**`task_struct`** (Case (1))	**`flags,uid,euid`**
`fork.c:610`	`0x134,138,13c`	**`task_struct`** (Case (1))	**`suid,fsuid,gid`**
`fork.c:610`	`0x140,144,148`	**`task_struct`** (Case (1))	**`egid,sgid,fsgid`**
`fork.c:610`	`0x1d0`	**`task_struct`** (Case (1))	**`cap_effective`**
`fork.c:610`	`0x1d4`	**`task_struct`** (Case (1))	**`cap_inheritable`**
`fork.c:610`	`0x1d8`	**`task_struct`** (Case (1))	**`cap_permitted`**
`generic.c:436`	`0x20`	**`proc_dir_entry`** (Case (2))	**`get_info`**
(Static object)		`proc_root_inode_operations`	`lookup`
(Static object)		`proc_root_operations`	`readdir`
(Static object)		`unix_dgram_ops`	`recvmsg`
(Module object)		`ext3_dir_operations`	`readdir`
(Module object)		`ext3_file_operations`	`write`

the latest one before the attack and the earliest one after the attack, and presents them for a comparison.

As a running example in this section, we will present the analysis of the attacks by the `adore-ng` rootkit. This rootkit is chosen because of its advanced malware behavior triggered by dynamic objects; and other rootkits can be analyzed in a similar way. Table 3 lists the kernel objects that the `adore-ng` rootkit tampers with. In particular, we focus on two specific attack cases using dynamic objects: (1) The first case is the manipulation of a PCB (T_3) for privilege escalation and (2) the second case is the manipulation of a function pointer in a dynamic `proc_dir_entry` object (P_1) to hijack kernel control flow. Fig. 4 presents a detailed view of kernel control flow and the usage of the targeted dynamic kernel memory in the attacks. The X axis shows the execution time, and kernel control flow is shown at top part of this figure. The space below shows the temporal usage of dynamic memory at the addresses of T_3 and P_1 before and after rootkit attacks. Thick horizontal lines represent the lifetime of kernel objects which are temporally allocated at such addresses. $+$ and $\times$ symbols below such lines show the read and write accesses on corresponding objects. The aforementioned analysis process is illustrated as solid arrows. From the times when T_3 and P_1 are manipulated (shown as dotted circles), the monitor scans the execution trace backward and forward to find the code execution that consumes the values read from such objects (i.e., $+$ symbols).

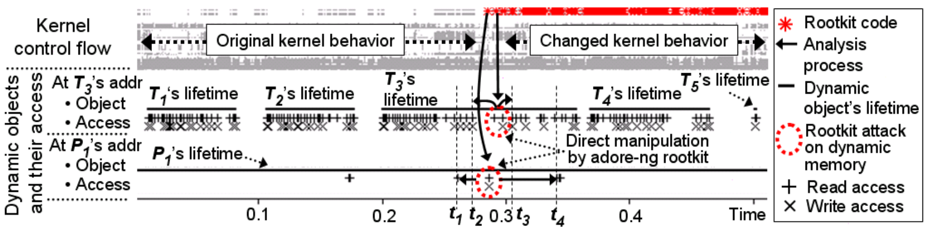
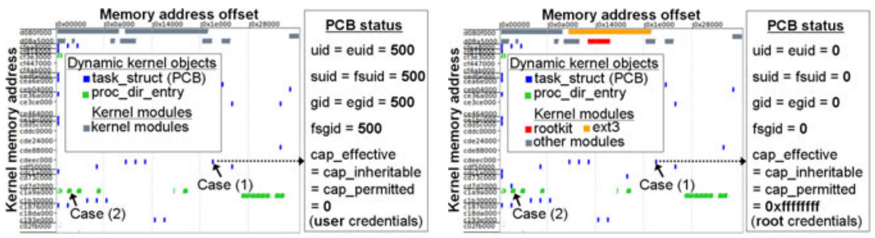


Fig. 4. Kernel control flow (top) and the usage of dynamic memory (below) at the addresses of T_3 (Case (1)) and P_1 (Case (2)) manipulated by the `adore-ng` rootkit. Time is in billions of kernel instructions.

Selecting semantically relevant kernel behavior using data lifetime. Our monitor inspects dynamic memory states in the temporal execution trace and as such we face the *dynamic data identity problem* described in Section 3.1. The core of the problem is that one memory address may correspond with multiple objects over a period of time. This problem can be solved if the lifetime of the inspected object is available because the monitor can filter out irrelevant kernel behaviors triggered by other kernel objects that share the same memory address. For example, in Fig. 4, we observe the memory for T_3 is used for four other PCBs (i.e., T_1, T_2, T_4, and T_5) as well in the history of kernel execution. Simply relying on the memory address to analyze the trace can lead to finding kernel behavior for *all five PCBs*. However, the monitor limits the inspected time range to the lifetime of T_3 and select only semantically relevant behaviors to T_3. Consequently it can provide a reliable inspection of runtime behavior only relevant to attacks.

Other kernel memory mapping approaches commonly cannot handle this problem properly. In static type-projection, when two kernel objects from different snapshots are given we cannot determine whether they represent the same data instance or not even though their status is identical because such objects may or may not be different data instances depending on whether memory allocation/deallocation events occur between the generation of such snapshots. Dynamic type-projection mapping is only based on malware instructions, and thus does not have information about allocation and deallocation events which occur during legitimate kernel execution.

Case (1): Privilege escalation using direct memory manipulation. In order to demonstrate the effectiveness of our temporal monitor we will discuss two specific attacks employed by `adore-ng`. The first is a privilege escalation attack that works by modifying the user and group ID fields of the PCB. The PCB is represented by T_3 in Fig. 4. To present the changed kernel behavior due to the manipulation of T_3, the temporal monitor finds the latest use of T_3 before the attack (at t_2) and the earliest use of it after the attack (at t_3). The data views at such times are presented in Fig. 5(a) and 5(b) as 2-dimensional memory maps where a kernel memory address is represented as the combination of the address in Y axis and the offset in X axis. These views present kernel objects relevant to this attack before and after the attack. The manipulated PCB is marked with "Case (1)" in the views and the values of its fields are shown in the box on the right side of each view (PCB status). These values reveal a stealthy rootkit behavior that changes the identity of

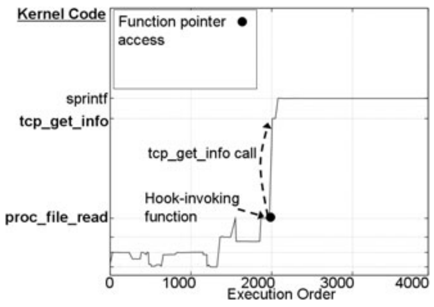


(a) The original data view at t_2. (b) The manipulated data view at t_3.

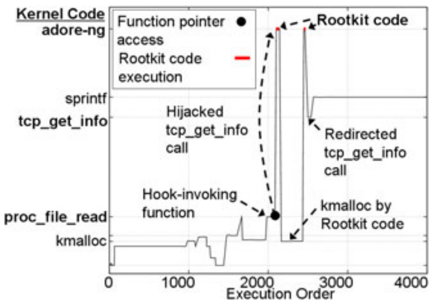
Fig. 5. Kernel data view before and after the `adore-ng` rootkit attack

a user process by directly patching its PCB (DKOM). Before the attack (Fig. 5(a)), the PCB has the credentials of an ordinary user whose user ID is 500. However, after the attack, Fig. 5(b) shows the credentials of the root user. This direct transition of its status between two accounts is abnormal in conventional operating system environments. `su` or `sudo` allow privileged operations by forking a process to retain the original identity. Hence we determine that this is a case of privilege escalation that illegally permits the root privilege to an ordinary user.

Case (2): Dynamic kernel object hooking. The next `adore-ng` attack hijacks kernel code execution by modifying a function pointer and this attack is referred to as Kernel Object Hooking (KOH) [10]. This behavior is observed when the influence of a manipulated function pointer in P_1 (see Fig. 4) is inspected. To select only the behaviors caused by this object, the monitor guides the analysis to the lifetime of P_1. The temporal monitor detects several behaviors caused by reading this object and two samples are chosen among those to illustrate the change of kernel behavior by comparison: the latest original behavior before the attack (at t_1) and the earliest changed behavior after the attack (at t_4). The monitor generates two kernel control flow graphs at these samples, each for a period of 4000 instructions. Fig. 6(a) and 6(b) present how this manipulated function pointer affects runtime kernel behavior. The Y axis presents kernel code; thus, the fluctuating graphs show various code executed at the corresponding time of X axis. A hook-invoking function (`proc_file_read`) reads the function pointer and calls the hook code pointed to by it. Before the rootkit attack, the control flow jumps to a legitimate kernel function `tcp_get_info` which calls `sprintf` after that as shown in Fig. 6(a). However, after the hook is hijacked, the control flow is redirected to the rootkit code which calls `kmalloc` to allocate its own memory, then comes back to the original function (Fig. 6(b)).



(a) The original control flow at t_1.



(b) The hijacked control flow at t_4.

Fig. 6. Kernel control flow view before and after the `adore-ng` rootkit attack

7 Discussion

Since LiveDM operates in the VMM beneath the hardware interface, we assume that kernel malware cannot directly access LiveDM code or data. However, it can exhibit potentially obfuscating behavior to confuse the view seen by LiveDM. Here we describe several scenarios in which malware can affect LiveDM and our counter-strategies to detect them.

First, malware can implement its own custom memory allocators to bypass LiveDM observation. This attack behavior can be detected based on the observation that any memory allocator must use internal kernel data structures to manage memory regions or its memory may be accidentally re-allocated by the legitimate memory allocator. Therefore, we can detect unverified memory allocations by comparing the resource usage described in the kernel data structures with the amount of memory being tracked by LiveDM. Any deviance may indicate the presence of a custom memory allocator.

In a different attack strategy, malware could manipulate valid kernel control flow and jump into the body of a memory allocator without entering the function from the beginning. This behavior can be detected by extending LiveDM to verify that the function was entered properly. For example, the VMM can set a flag when a memory allocation function is entered and verify the flag before the function returns by interceding before the return instruction(s) of the function. If the flag was not set prior to the check, the VMM detects a suspicious memory allocation.

8 Related Work

Static type-projection mapping has been widely used in the defense against kernel malware attacks. SBCFI [16] detects persistent manipulations to the kernel control flow graph by using kernel memory maps. Gibraltar [1] derives data invariants based on a kernel memory map to detect kernel malware. KOP [5] improves the accuracy of mapping using extended type graph based on static analysis in addition to memory analysis. Complementing these approaches, allocation-driven mapping provides an un-tampered view of kernel objects where their identification is not affected by kernel malware's manipulation of the kernel memory content. It also accurately reflects the temporal status of dynamic kernel memory, which makes it applicable to temporal analysis of kernel/kernel malware execution.

PoKeR [23] and Rkprofiler [31] use dynamic type-projection mapping generated from rootkit instructions to understand the rootkit behavior. Since only rootkit activity is used as the input to generate a kernel memory map, this approach can only cover the kernel objects directly manipulated by rootkit code. Moreover, there exist the attacks that are difficult to be analyzed by these profilers because rootkits can use various resource such as hardware registers to find the attack targets [21].

KernelGuard (KG) [20] is a system that prevents DKOM-based kernel rootkits by monitoring and shepherding kernel memory accesses. It identifies kernel objects to be monitored by scanning the kernel memory using data structure-specific policies enforced at the VMM level. Similar to type-projection mapping, KG's view of kernel memory is based on the runtime kernel memory content which is subject to malware manipulation. As such, KG's reliability can be improved by adopting LiveDM as the underlying kernel memory mapping mechanism.

LiveDM involves techniques to capture the location, type, and lifetime of individual dynamic kernel objects, which can be described as belonging to the area of virtual machine introspection [9].

9 Conclusion

We have presented allocation-driven mapping, a kernel memory mapping scheme, and LiveDM, its implementation. By capturing the kernel objects' allocation and deallocation events, our scheme provides an un-tampered view of kernel objects that will not be affected by kernel malware's manipulation of kernel memory content. The LiveDM-generated kernel object map accurately reflects the status of dynamic kernel memory and tracks the lifetimes of all dynamic kernel objects. This temporal property is highly desirable in temporal kernel execution analysis where both kernel control flow and dynamic memory status can be analyzed in an integrated fashion. We demonstrate the effectiveness of the LiveDM system by developing a hidden kernel object detector and a temporal malware behavior monitor and applying them to a corpus of kernel rootkits.

Acknowledgements. We thank the anonymous reviewers for their insightful comments. This research was supported, in part, by the Air Force Research Laboratory (AFRL) under contract FA8750-09-1-0224 and by the National Science Foundation (NSF) under grants 0716444, 0852131, 0855036 and 0855141. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the AFRL or NSF.

References

1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), pp. 77–86 (2008)
2. Bellard, F.: QEMU: A Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
3. Boehm, H.J., Weiser, M.: Garbage Collection in an Uncooperative Environment. Software, Practice and Experience (1988)
4. Butler, J.: DKOM (Direct Kernel Object Manipulation), `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009 (2009)
6. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In: Proceedings of 2008 USENIX Annual Technical Conference, USENIX 2008 (2008)
7. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging For Data Structures. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
8. Free Software Foundation: The GNU Compiler Collection, `http://gcc.gnu.org/`
9. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, NDSS 2003 (2003)
10. Hoglund, G.: Kernel Object Hooking Rootkits (KOH Rootkits), `http://www.rootkit.com/newsread.php?newsid=501`
11. Hund, R., Holz, T., Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: Proceedings for the 18th USENIX Security Symposium (2009)

12. Lin, Z., Riley, R.D., Xu, D.: Polymorphing Software by Randomizing Data Structure Layout. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 107–126. Springer, Heidelberg (2009)
13. MITRE Corp.: Common Vulnerabilities and Exposures, `http://cve.mitre.org/`
14. Parallels: Parallels, `http://www.parallels.com/`
15. Petroni, N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings for the 13th USENIX Security Symposium (August 2004)
16. Petroni, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007 (2007)
17. Petroni, N.L., Walters, A., Fraser, T., Arbaugh, W.A.: FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory. Digital Investigation Journal 3(4), 197–210 (2006)
18. Petroni, Jr. N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the 15th Conference on USENIX Security Symposium, USENIX-SS 2006 (2006)
19. Polishchuk, M., Liblit, B., Schulze, C.W.: Dynamic Heap Type Inference for Program Understanding and Debugging. In: Proceedings of the 34th Annual Symposium on Principles of Programming Languages. ACM, New York (2007)
20. Rhee, J., Riley, R., Xu, D., Jiang, X.: Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In: International Conference on Availability, Reliability and Security, ARES 2009 (2009)
21. Rhee, J., Xu, D.: LiveDM: Temporal Mapping of Dynamic Kernel Memory for Dynamic Kernel Malware Analysis and Debugging. Tech. Rep. 2010-02, CERIAS (2010)
22. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
23. Riley, R., Jiang, X., Xu, D.: Multi-Aspect Profiling of Kernel Rootkit Behavior. In: Proceedings of the 4th European Conference on Computer Systems (Eurosys 2009) (April 2009)
24. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of 21st Symposium on Operating Systems Principles (SOSP 2007). ACM, New York (2007)
25. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007), pp. 552–561. ACM, New York (2007)
26. Sun Microsystems, Inc: VirtualBox, `http://www.virtualbox.org/`
27. The Month of Kernel Bugs archive, `http://projects.info-pull.com/mokb/`
28. US-CERT: Vulnerability Notes Database, `http://www.kb.cert.org/vuls/`
29. VMware, Inc.: VMware Virtual Machine Technology, `http://www.vmware.com/`
30. Wei, J., Payne, B.D., Giffin, J., Pu, C.: Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense. In: Proceedings of the 24th Annual Computer Security Applications Conference, ACSAC 2008 (December 2008)
31. Xuan, C., Copeland, J.A., Beyah, R.A.: Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In: Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009), pp. 304–325 (2009)