CERIAS Tech Report 2010-02

LiveDM: Temporal Mapping of Dynamic Kernel Memory for Dynamic Kernel Malware Analysis and Debugging

by Junghwan Rhee, Dongyan Xu
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

LiveDM: Temporal Mapping of Dynamic Kernel Memory for Dynamic Kernel Malware Analysis and Debugging

Junghwan Rhee Dongyan Xu
Department of Computer Science, Purdue University
{rhee,dxu}@cs.purdue.edu

Abstract

Dynamic kernel memory is difficult to analyze due to its volatile status; numerous kernel objects are frequently allocated or freed in a kernel's heap, and their data types are missing in the memory systems of current commodity operating systems. Since the majority of kernel data is stored dynamically, this memory has been a favorite target of many malicious software and kernel bugs. In order to analyze dynamic kernel memory, a global technique that systematically translates a given memory address into a data type is essential.

Previous approaches had a limited focus in the analysis of either a malware's execution or a snapshot of kernel memory. We present here a new memory interpretation system called LiveDM that can automatically translate dynamic kernel memory addresses into data types.[1] *This system enables the accurate memory analysis of the entire kernel execution, ranging from malware activity to legitimate kernel code execution, over a period of time beyond the instant of a snapshot by using these two novel techniques. (1) The system identifies an individual dynamic kernel object with its systematically-determined runtime identifier that points to the code where the object is allocated. (2) The data type then can be automatically extracted from the code using static code analysis offline.*

We have implemented a prototype of LiveDM that supports three Linux kernels where LiveDM dynamically tracks tens of thousands of dynamic kernel memory objects that can be accurately translated into data types in the offline process. We have evaluated and validated its general applicability and effectiveness in extensive case studies of kernel malware analysis and kernel debugging.

1 Introduction

Dynamic memory is an essential but complicated mechanism to handle data in programs. In user programs such memory is placed in the heap, and the heap memory is notorious for segmentation fault errors. In spite of today's state-of-the-art debugging technology, it is difficult to understand this memory due to the lack of accurate information as to the locations and data types of dynamic memory objects.

This situation can worsen in the kernel side, where higher reliability is expected but less monitoring and debugging support is provided compared to the user space. Operating system kernels manage the majority of kernel data in dynamic memory, thus many security or reliability issues begin there. For instance, an increasing number of kernel malware programs target dynamic kernel objects [1, 2, 3, 4, 5], and many kernel bugs are caused by dynamic memory errors [6, 7, 8].

Accurate analysis of dynamic kernel memory faces the following challenges. Operating system kernels frequently allocate or deallocate numerous dynamic objects. Hence, maintaining an accurate view of dynamic kernel

[1] LiveDM is the acronym for the **Live D**ynamic kernel memory **M**ap.

objects requires fine-grained capturing of memory allocation activities. Furthermore, a mechanism is needed to systematically identify the types of objects because current kernel memory systems do not store type information for dynamic objects. A type-enabled kernel memory tracking system will provide an improved semantic view that illustrates how the kernel heap is organized and explains the purpose of an individual dynamic object, thereby greatly assisting the investigators of kernel attacks or bugs.

Two approaches generally can be utilized to implement such a system. If current memory systems can be extended to support types [9, 10], not only the memory management code will be modified but also all allocation code will require changes to assign a type for each allocation (recall that most current kernel memory allocators obtain a size but not a type). The other approach, which we will shortly present, is applicable to commodity operating system kernels without any change by tracking the type information via a virtual machine monitor and offline analysis.

Previous approaches could infer dynamic data types using Bayesian unsupervised learning [11] or point-to analysis that resolves generic pointers and type ambiguity [12]; but their analyses were focused on the instant when the snapshot was taken so their use was limited to deriving signatures of programs or analyzing persistent memory manipulation. These approaches cannot be applied to analyze runtime kernel execution where temporal memory errors (e.g., buffer overflow) occur or to investigate sophisticated kernel attacks that avoid persistent manipulation (e.g., DKOM [2, 5]).

Another closely related work is PoKeR [4], a kernel rootkit profiler, which traces a rootkit's instructions and translates the manipulated memory targets under assumptions about the rootkit's attack behavior. However, as we will show in Section 4.3, such assumptions can be violated to elude PoKeR. In addition, the analysis focus of PoKeR is limited to a number of kernel objects manipulated by kernel rootkits, thus this technique cannot be applied to inspect ordinary dynamic kernel objects accessed by legitimate kernel code.

In this paper we will present LiveDM, a new dynamic kernel memory interpretation system that addresses these problems. LiveDM overcomes the limitations of previous approaches by keeping track of the address range of an individual dynamic object and its allocation code position from which its type is derived. The contributions of this system can be summarized as follows.

- LiveDM systematically identifies an individual dynamic kernel object with its runtime identifier pointing to the code where the object is allocated. This object's type can be extracted from the designated code offline. Since LiveDM handles dynamic kernel memory by capturing the activity of kernel memory allocators, this approach extends the coverage of memory analysis to the dynamic kernel objects accessed by *the entire kernel execution* beyond the scope of dynamic memory targeted by kernel rootkits [4].

- In order to ensure the correct analysis of the volatile dynamic kernel memory's status, LiveDM updates the kernel memory map in the virtual machine monitor (VMM) whenever a memory allocation and deallocation occurs in the guest kernel. This mechanism enables the accurate translation of dynamically changing memory *over a period of time*, which is not possible in snapshot-based approaches [11, 12] unless a snapshot is taken for each memory access.

- In the offline process, a runtime identifier for each dynamic memory object can be automatically translated into a data type by using *static code analysis*. Specifically, the memory type is captured from the source code that handles the allocated memory. By following the dependencies among the internal representations of kernel code generated by compilers, the type information can be automatically determined.

We have implemented a prototype of LiveDM that can translate the dynamic objects of three off-the-shelf Linux distributions and have evaluated it in the analysis of malware and kernel bugs to show its general applicability. LiveDM targets the environments that investigate internal kernel activities, such as honeypots or kernel debugging setups, that can trade an incremental performance overhead for a new in-depth analysis capability of dynamic kernel memory.

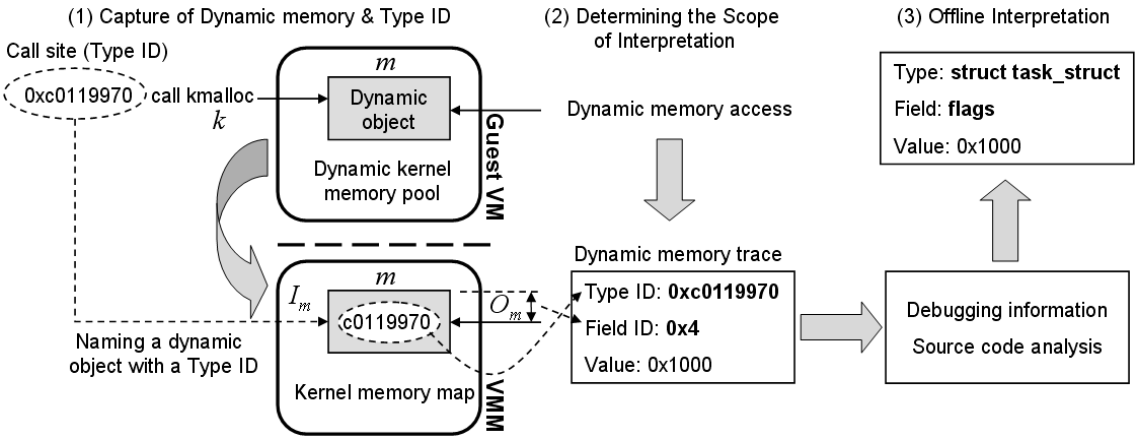


Figure 1. Overview of LiveDM.

2 Design of LiveDM

In this section we will first introduce our approach to determine the data types of dynamic memory objects, then we will present the specific techniques of LiveDM in three phases.

2.1 Call-site-based Dynamic Data Type Identification

LiveDM determines the data type of a dynamic kernel object by using its *allocation code*. When a new memory is allocated in the kernel, the allocation code can be designated by the *call site* of a memory allocation function.[2] By recording this call site as a runtime identifier, we can enable a dynamic kernel memory object to point to the code used to derive its type. Once the code is identified, the type can be extracted by traversing the code. For instance, if a newly allocated memory address is assigned to a variable, the type of this variable is extracted because it can represent the type of the allocated object. In a correctly compiled program, such as a running kernel, a declaration and its type definition must precede the use of the variable since the dependencies among those code elements are checked during the compilation. By instrumenting the compilers, we can follow such dependencies and identify the type of the allocated memory. In this paper we call this identifier a type ID.

Definition (Type ID)**.** *Given a dynamic memory object m, a type ID, denoted as I_m, is the call site of the dynamic memory allocation function k executed to allocate m. In other words, I_m is the program counter of the call instruction that invokes k to allocate m.*

Figure 1 illustrates a high level view of our approach in three phases. In the first phase, for a newly allocated dynamic kernel object in a guest kernel, its type ID is systematically determined. Its address range and type ID are recorded in a shadow memory space in VMM called *the kernel memory map*. LiveDM uses this map for three purposes. First, for a given address, LiveDM determines whether it is for dynamic memory by looking up the designated object using the address. Second, LiveDM maintains type IDs for kernel objects transparently to the guest kernel by storing them in this map; and finally this map is used for a more specific interpretation to a field within an object as described below.

A dynamic memory is often used as an instantiation of a composite data type (i.e., non-primitive type such as `struct`) that has fields, and it will be informative to point to a specific field in the interpretation in such cases. In the low level view, a field is represented as an offset in the chunk of memory used for the data structure. The kernel memory map provides the address range of an object, from which we can determine the offset within the

[2] A call site for a function is the program counter of a call instruction for the function.

object, in order to identify a field during runtime. This identification is interpreted into a field name in the offline analysis. In this paper, we define this identifier as follows.

Definition (Field ID)**.** *Given a memory address i that belongs to the address range of a dynamic memory object m, a field ID of i, is the offset within the range.*

The second phase shows how a type ID and a field ID are determined for an inspected dynamic memory address, which are translated to a data type and a field in the offline interpretation. In the final phase, first we convert a type ID into the position of the corresponding code by using debugging information. Using the instrumented compiler, we then traverse the code structures and derive the data type of the object for the given address. In the following sub-sections, we present the techniques for each phase in detail.

2.2 Phase 1: Transparent Capture of Dynamic Objects and Runtime Identification

Before runtime monitoring begins, we instruct the VMM which kernel memory allocation/deallocation functions to intercept. The functions can be identified by using debugging information and the kernel symbol table. In Section 3, the implementation details such as the scope of the captured memory functions and the handling of wrapper functions will be presented. When kernel code is about to run, the VMM intercepts the control with the fetched address. If the code matches either an allocation function or a deallocation function, a respective VMM capture function is called. The allocated memory range can be expressed as the initial address and the size, and such information can be obtained from a memory allocation function call. The initial address is the return value of the function and the size is given as a parameter. In order to free a memory range, the initial address is sufficient information to search the block to be freed, and it is given as a parameter of the deallocation function call.

Leveraging the standard rules about the delivery of such values, called *function call conventions*, LiveDM can capture these memory operations without modifying a guest kernel code.[3] Function parameters are delivered through the stack or registers, and we can capture them by inspecting their locations at the callee. The return value can be captured by determining the location and the time when it is passed. The location is determined by the function call convention. The integers up to 32 bits and the 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 function returns to the caller, and this moment can be recognized using the return address extracted at the callee. The VMM stores such return addresses in the shadow stack. Then when the code that the CPU is about to execute is matched with the code in this stack, the VMM captures the return value from the `EAX` register. LiveDM does this match by searching the stack with the address in LIFO order because the return sequence of kernel functions may not be the exact reverse order of the call sequence due to nondeterministic execution of multiple contexts.

There can be multiple approaches to capture a type ID (a call site of a function call) such as instrumenting call instructions in VMM. LiveDM approximates the call site with the return address of a function call and captures it when a kernel memory function is about to run. This approximation is used mainly to simplify the implementation and to minimize the interception. The captured address is the next address of the call site in the instruction stream. In the source code analysis, this approximated call site will point to the same or the next non-comment line of the allocation call. This trivial offset can be easily handled in the offline analysis procedure. The captured type ID is stored in the kernel memory map along with the memory address range.

2.3 Phase 2: Determining the Scope of Interpretation

The previous phase introduced the technique to track down the types of the dynamic objects. In the application scenarios, we will collect a set of memory addresses to be investigated. There are generally two ways to set the scope of the kernel memory addresses to be interpreted.

[3] A function call convention is a scheme to pass function parameters and a return value. We use the ones for x86 architecture and GNU Compiler Collection (`gcc`) compilers [13].

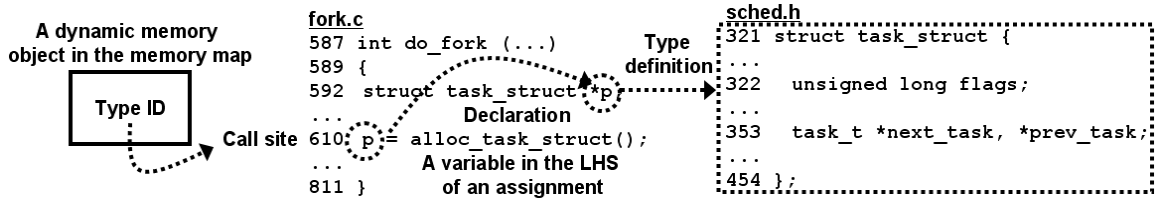
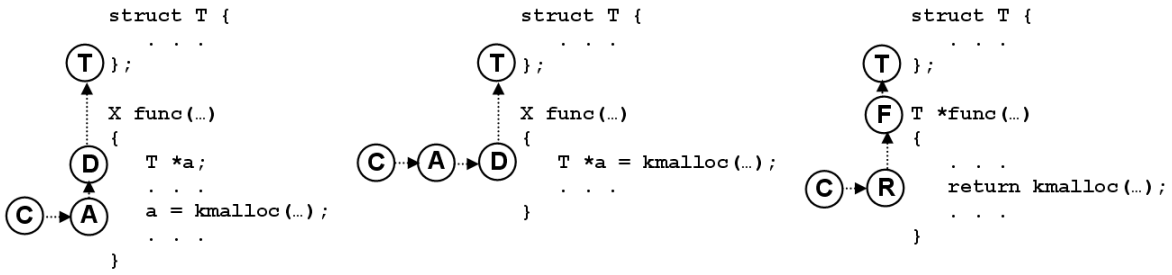


Figure 2. A high level view of static code analysis



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

Figure 3. Cases of analyzed code. C: a call site, A: an assignment, D: a variable declaration, T: a type definition, R: a return, and F: a function declaration.

First, if the diagnosis of the entire dynamic kernel memory status is desired, a snapshot of the kernel memory map can be taken to interpret the whole. Unlike a conventional memory dump [14], this snapshot has the address ranges of dynamic kernel objects and their type IDs to derive their types. Therefore, it provides significantly improved semantic information to understand kernel memory status. Snapshot-based approaches [12] infer memory graphs by following pointers and mapping the dereferenced memory addresses to pointers' types. Our approach construct the kernel memory map not by using contents (pointer values) of memory but by using allocation events, thus it is more tolerant to invalid pointer addresses or memory casting to generic pointer types.

Second, the scope of instructions can be set to trace the list of dynamic memory objects accessed by kernel memory instructions. This mechanism enables the diagnosis of dynamically changing memory status over a period of time with accurate type information. Because the dynamic status can significantly change in any moment, the snapshot-based approaches can only achieve similar accuracy by making a snapshot for an individual memory access, which requires significant CPU and storage overhead. In order to define the tracing focus with policies, LiveDM has a memory address space called *Trace map*, which is checked for each code fetch to select the instruction to be traced. We can also define complicated taint rules to track a malware's activity based on its code and the overwritten memory. In Section 5, we will present case studies using this feature and the tracing policies are respectively defined.

When a memory access is traced, the identification of the accessed dynamic object is retrieved from the kernel memory map and logged to reflect the accurate dynamic memory status at runtime. If a dynamic memory object corresponding to the address is found, its type ID is retrieved and the field ID is calculated as the offset within the address range. The kernel memory map is built with a page table structure and a hash table for efficient lookups.

2.4 Phase 3: Offline Data Type Interpretation

In this step, the kernel object type IDs collected in the previous step are translated into data types via static kernel code analysis. The intuition behind this technique is that the type of a dynamic object can be found from

the source code that handles the object's allocation. Figure 2 illustrates a high level view of our mechanism. First, the type ID (allocation call site) (C) of a dynamic object is mapped to the source code `fork.c:610` using debugging information. This code assigns the allocated memory to a pointer variable at the left-hand side (LHS) of the assignment (A). In this case, this variable's type can represent the type of the allocated memory. Thus, the declaration of this pointer (D) and the definition of its type (T) are consequently searched by traversing the code. Specifically, during the compilation, the parser sets the dependencies among the internal representations (IRs) of such code elements; therefore, the type can be found by generating the IRs and following their dependencies.

For object type resolution, there are various patterns in the allocation code as shown in Figure 3. Case 1 is the typical pattern (C→A→D→T) that we previously explained. `gcc` recognizes the pattern of Case 2 differently as an initialized declaration. However, the instrumented code generates the IRs for both of an assignment and a declaration at the same line, thereby treating this case similar to the first case. Unlike the first two cases, the third pattern 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 Figure 3(c), this pattern is presented as C→R→F→T.

Prior to static code analysis, we generate the sets of information about the code elements to be traversed by compiling the kernel source code with the `gcc` compiler [13] that we instrumented. This compiler generates several internal representations for the compiled code, such as Abstract Syntax Tree (AST) and Register-Transfer Language (RTL). We choose AST because the type information, such as the definitions of composite types, is available in this representation.

3 Implementation

In our prototype, LiveDM supports three off-the-shelf Linux operating systems of different kernel versions: Fedora Core 6 (Linux 2 6.18), Debian Sarge (Linux 2.6.8), and Redhat 8 (Linux 2.4.18). Our mechanism is general enough to work with any operating system that follows the standard function call conventions. LiveDM can be easily implemented on any software virtualization system, such as VMware (Workstation, Server, and Player) [15], VirtualBox [16], and Parallels [17]. We chose QEMU virtual machine 0.9.0 [18] with the KQEMU optimizer for implementation convenience. We note that the QEMU-based implementation with relatively high performance overhead can indirectly benefit production workloads running on another high-performance VMM by taking the outsourced replay approach such as the decoupled analysis in [19].

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 the preprocessor; thus, their call sites are captured in the same way as core functions. However, in the case of allocations through regular wrapper functions, the call sites will belong to the wrapper code.

In order to solve this problem, we take two approaches. If a wrapper is locally used only in a few code, 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 such wrappers as memory allocation functions. Operating systems, which have a 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 categorized 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 general, we captured about 20 functions in each guest kernel to capture the dynamic memory ranges.

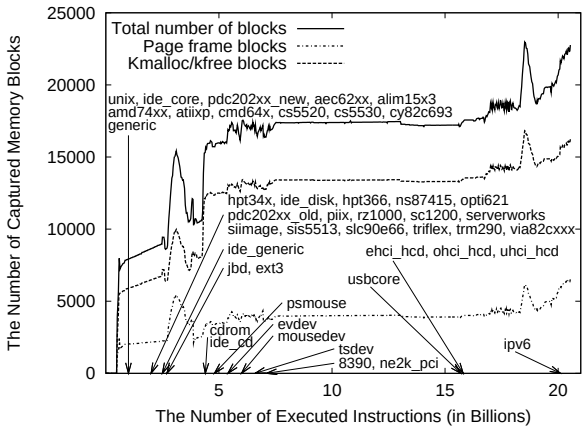


Figure 4. The usage of dynamic kernel objects during the booting stage (OS: Debian Sarge).

The memory functions in an OS can be determined from the design specification (e.g., the Linux Kernel API) or 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 operating systems, 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 code with the distributed kernels, but the code offsets can be slightly different due to the additional debugging information.

For static analysis, we compiled the source code of the experimented kernels using the `gcc` [13] compiler (version 3.2.3) that we instrumented. We placed hooks in the parser and extracted the ASTs for the code elements necessary in the static code analysis, which are described in Section 2.4.

4 Evaluation

We first evaluate the core properties of LiveDM especially its accuracy and robustness against stealthy malware. We then present a number of case studies that use LiveDM for kernel malware analysis in Section 5. The guest systems are configured with 256MB RAM and the host machine has a 3.2Ghz Pentium D CPU and 2GB RAM.

4.1 Identifying Dynamic Kernel Objects

In order to evaluate the core feature of LiveDM that captures dynamic objects, we measured the number of active dynamic kernel objects over the booting process. The total number of valid objects varied over the booting period as shown in Figure 4. After the system was fully booted, LiveDM for Debian Sarge was tracking 22765 dynamic kernel memory blocks. In Figure 4, the number of blocks allocated by the `kmem_cache_alloc/free` functions and the `kmalloc/kfree` functions were merged because `kmalloc/kfree` functions call `kmem_cache_alloc/free` functions internally and were therefore considered to be a similar kind. There were a limited number of `vmalloc` objects, which were mainly used for kernel modules. So we denoted the names of loaded kernel modules instead of depicting their numbers.

We present a list of core kernel data structures that LiveDM captures at runtime in Table 1. These data structures manage the core operating system status such as process information, memory mapping of each process, and the

	Type ID (A/R)	Declarations (D/F)	Case	Data Type	#Objects
Task/Signal	kernel/fork.c:248	kernel/fork.c:243	1	task_struct	66
	kernel/fork.c:801	kernel/fork.c:795	1	sighand_struct	63
	fs/exec.c:601	fs/exec.c:587	1	sighand_struct	1
	kernel/fork.c:819	kernel/fork.c:813	1	signal_struct	66
Memory	arch/i386/mm/pgtable.c:229	arch/i385/mm/pgtable.c:229	2	pgd_t	54
	kernel/fork.c:433	kernel/fork.c:431	1	mm_struct	47
	kernel/fork.c:559	kernel/fork.c:526	1	mm_struct	7
	kernel/fork.c:314	kernel/fork.c:271	1	vm_area_struct	149
	mm/mmap.c:923	mm/mmap.c:748	1	vm_area_struct	1004
	mm/mmap.c:1526	mm/mmap.c:1521	1	vm_area_struct	5
	mm/mmap.c:1722	mm/mmap.c:1657	1	vm_area_struct	48
	fs/exec.c:402	fs/exec.c:342	1	vm_area_struct	47
File system	kernel/fork.c:677	kernel/fork.c:654	1	files_struct	54
	kernel/fork.c:597	kernel/fork.c:597	2	fs_struct	53
	fs/file_table.c:76	fs/file_table.c:69	1	file	531
	fs/buffer.c:3062	fs/buffer.c:3062	2	buffer_head	828
	fs/block_dev.c:233	fs/block_dev.c:233	2	bdev_inode	5
	fs/dcache.c:692	fs/dcache.c:689	1	dentry	4203
	fs/inode.c:112	fs/inode.c:107	1	inode	1209
	fs/namespace.c:55	fs/namespace.c:55	2	vfsmount	16
	fs/proc/inode.c:93	fs/proc/inode.c:90	1	proc_inode	237
	drivers/block/ll_rw_blk.c:1405	drivers/block/ll_rw_blk.c:1405	2	request_queue_t	18
	drivers/block/ll_rw_blk.c:2950	drivers/block/ll_rw_blk.c:2945	1	io_context	10
Network	net/socket.c:279	net/socket.c:278	1	socket_alloc	12
	net/core/sock.c:617	net/core/sock.c:613	1	sock	3
	net/core/dst.c:125	net/core.dst.c:119	1	dst_entry	5
	net/core/neighbour.c:265	net/core/neighbour.c:254	1	neighbour	1
	net/ipv4/tcp_ipv4.c:134	net/ipv4/tcp_ipv4.c:133	2	tcp_bind_bucket	4
	net/ipv4/fib_hash.c:586	net/ipv4/fib_hash.c:461	1	fib_node	9

Table 1. Dynamic kernel objects identified by using allocation code (Type ID) and related code elements (Declarations). A/R: an assignment or a return, D/F: a variable declaration or a function declaration. (OS: Debian Sarge).

status of file systems and network which are often targeted by kernel malware and kernel bugs [4, 20, 21, 22, 1, 8, 6, 7]. Kernel objects are recognized using runtime identifiers in column **Type ID** during runtime. In offline, these IDs are translated into data types shown in column **Data Type** by traversing the allocation code and the declarations (shown in column **Declarations**) used for the allocation in the kernel source code. Column **Case** shows the case of allocation code in static analysis presented in Section 2.4. The numbers of identified objects are presented in column **#Objects**.

4.2 The Accuracy of LiveDM

We evaluate the accuracy of LiveDM in kernel object type resolution. For experimental purposes, we instrument the kernel memory functions described in Section 3 and generate a log of allocation and deallocation events of dynamic kernel objects. We observed that the active dynamic objects derived from such events accurately match the live dynamic kernel objects systematically captured by LiveDM using virtual machine techniques.

The type derivation accuracy is checked by traversing the kernel source code and translating the call sites from the instrumented code as done by related approaches [12, 11]. The derived types at the allocation code are completely matched with the results from our automatic static code analysis technique.

```
if (__this_module.next)
    __this_module.next = __this_module.next->next;
```

Figure 5. The cleaner rootkit of the adore-ng rootkit distribution can void PoKeR's profiling.

```
unsigned int * pgd;
__asm__("movl %%cr3,%0":"=r" (pgd));
pgd = __va(pgd) + pmd_offset;
(*pgd) = manipulated_pmd;
```

Figure 6. REGIKIT: A rootkit that obtains the attack target from a hardware register.

```
struct dentry *adore_lookup(struct inode *i,
    struct dentry *d, struct nameidata *nd)
{
  struct task_struct *tsk;
  if (strncmp(d->d_iname, "pr-", 3) == 0) {
    tsk = find_task_by_pid(adore_atoi(d->d_iname + 3));
    tsk->uid = tsk->suid = tsk->euid = tsk->fsuid = \
    tsk->gid = tsk->egid = tsk->fsgid = 0;
    tsk->cap_effective = tsk->cap_inheritable = \
    tsk->cap_permitted = ~0UL;
  }
}
```

Figure 7. A modified adore-ng for Linux 2.6. The targeted process is assumed to be a temporarily suspended user shell that is forked before this rootkit is loaded.

4.3 Profiling Resistant Attacks

In this section, we demonstrate the robustness of LiveDM against stealthy rootkits that can elude an existing kernel malware profiler, PoKeR [4]. PoKeR assumes the attack behavior that (1) starts scanning static objects and proceeds following pointers until the intended target is found. PoKeR identifies the targets by following the accessed memory in the same way as rootkits. The data types of static objects are known, and the types of subsequently accessed objects can be inferred using the dereferenced pointer types. In addition, once PoKeR is activated, (2) it then tracks the addresses of the scheduled PCBs.[4] Thus the attack target on the current PCB can be identified even though the rootkit's behavior does not follow the first assumption. However, rootkits may well violate these assumptions. We found that at least two existing rootkits can avoid being profiled by PoKeR and more techniques to elude PoKoR are presented below.

Using Dynamic Module Symbols: Figure 5 presents the attack code of the cleaner rootkit included in the adore-ng rootkit distribution. In the code, this rootkit locates the current module structure using the `__this_module` symbol. This symbol is a dynamic module symbol locally defined for each kernel module; the kernel module loader dynamically maps the current module's address to this symbol when the module is loaded. Since neither this symbol is reached from static objects nor the manipulated memory is part of scheduled PCBs, both of PoKeR's assumptions are not followed; thus PoKeR cannot identify this manipulation.

Using Registers: This technique violates the first assumption by obtaining the attack target directly from a hardware register. For example, the `CR3` register is used to load a page table directory and a page table directory can be manipulated using this register as shown in Figure 6. As another example, the modhide rootkit in the knark rootkit distribution uses the `EBX` register to locate the current module structure in Linux 2.2 kernels. Similarly, any hardware register (other than the `ESP` register that points to a scheduled PCB) can be used to find the attack target that can evade PoKeR.

Using Kernel Functions: PoKeR can identify types of dynamic objects based on the way that a rootkit manipulates it, so the other objects that are used by legitimate kernel execution are not understood by PoKeR. For instance, in Linux 2.6 kernels there is a function, `find_task_by_pid` that returns the address of the PCB for a given process identification number (PID). Any other function that returns the address of a dynamic object likewise can be used to elude PoKeR.

Turning a Real World Rootkit into a PoKeR Resistant Rootkit: By applying the presented techniques

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

Rootkit Name	Runtime Identification		Offline Interpretation		Operating System (OS)
	Type ID	Field ID	Type	Field/Offset	
cleaner	kernel/module.c:314	0x4	module	next	RedHat 8 (Linux 2.4)
modhide	kernel/module.c:314	0x4	module	next	
REGIKIT	arch/i386/mm/pgtable.c:229	offset pmd_offset	pgd_t	offset pmd_offset	Debian Sarge (Linux 2.6)
PoKeR resistant adore-ng	kernel/fork.c:248	0x1d0,1d4,1d8,1dc	task_struct	uid,euid,suid,fsuid	
	kernel/fork.c:248	0x1e0,1e4,1ec	task_struct	gid,eguid,fsgid	
	kernel/fork.c:248	0x1f4	task_struct	cap_effective	
	kernel/fork.c:248	0x1f8	task_struct	cap_inheritable	
	kernel/fork.c:248	0x1fc	task_struct	cap_permitted	

Table 2. The dynamic kernel objects manipulated by PoKeR resistant rootkits.

together, we can turn an existing kernel rootkit into a *PoKeR resistant rootkit* in which the targeted dynamic memory cannot be understood by PoKeR. For instance, we can make adore_ng rootkit resistant to PoKeR. One of the functions of this rootkit, `adore_lookup`, is invoked by a lookup on the `proc` file system. By taking a command as a directory name, this function works as a backdoor that controls the kernel. The original code gives the root privilege to the current user by manipulating the current user's PCB. Instead, we use `find_task_by_pid` to look up a non-current process with a given PID. In particular, this modified rootkit assumes to handle a suspended user shell that is forked before the rootkit is loaded. The combination of these techniques therefore violates both of the assumptions by PoKeR. The attacker can trigger the `adore_lookup` with the PID of the sleeping shell. Then the rootkit sets its PCB with the root credentials. The attacker can wake up this shell any time to become a root.

Analyzing PoKeR Resistant Rootkits: Unlike PoKeR, LiveDM has no assumption regarding how the manipulated address should be obtained. Therefore, it can interpret an address regardless of the rootkit behavior that finds the target. Table 2 summarizes the attack targets of the PoKeR resistant rootkits identified by LiveDM. First, we can confirm that the cleaner and modhide rootkits both manipulate the module structures. The modhide rootkit is written for Linux 2.2 kernels so we slightly modified it to use the `EBP` register instead of the `EBX` in order to run it in the Redhat 8 system (Linux 2.4). LiveDM identified that the REGIKIT rootkit manipulates a dynamic object of `pgd_t` type. A page table directory is a dynamic object of this type, thereby confirming the attack target. The memory manipulated by the modified adore-ng matches the data targeted by the attack code shown in Figure 7. The manipulated object is a PCB (of `task_struct` type) and specific fields under attack are also matched to the code.

5 Case Studies

LiveDM provides new aspects of kernel memory analysis by interpreting dynamic kernel memory addresses into data types. In this section, we first will present an attack case that user program code overwrites kernel memory by triggering kernel bugs. Since LiveDM manages the kernel memory map, it can interpret the manipulated memory just based on memory instructions that overwrite kernel memory. Next, we will present the attack targets by real world kernel rootkits, in particular on dynamic kernel memory using LiveDM's memory interpretation. We then will present a new aspect of kernel rootkit behavior that allocates the rootkit's own memory, revealing how rootkits use this flexible runtime storage to store attack code or their own data. Finally, we broaden our application to the memory analysis of kernel bugs. By using LiveDM's type translation, we can obtain the types of dynamic kernel objects accessed by kernel code statements; therefore we can identify kernel memory accesses to abnormal kernel objects triggered by kernel bugs.

5.1 User Level Root Exploit Attack Analysis

Vmsplice root exploit (CVE-2008-0009, CVE-2008-0010, CVE-2008-0600) is a notorious user level attack leveraging critical bugs of recent Linux kernels. It allowed an ordinary user to easily obtain the root privilege by running a simple proof-of-concept code widely available. The effected kernels span on multiple kernel versions

Type ID	Field ID	Data Type	Field	Value
kernel/fork.c:164	0x158,154,150,14c	task_struct	fsuid,suid,euid,uid	0x0
kernel/fork.c:164	0x168,164,160,15c	task_struct	fsgid,sgid,egid,gid	0x0
kernel/fork.c:164	0x178	task_struct	cap_permitted	0xffffffff
kernel/fork.c:164	0x174	task_struct	cap_inheritable	0xffffffff
kernel/fork.c:164	0x170	task_struct	cap_effective	0xffffffff

Table 3. Kernel memory victims overwritten by vmsplice root exploit attack. (OS: Fedora Core 6)

```
void kernel_code()
{
 int i;
 uint *p = get_current();             /* The pointer of current task_struct is obtained */
                                      /* from the ESP register.                         */
 for (i = 0; i < 1024-13; i++) {
   if (p[0] == uid && p[1] == uid &&
       p[2] == uid && p[3] == uid &&
       p[4] == gid && p[5] == gid &&
       p[6] == gid && p[7] == gid) {
    p[0] = p[1] = p[2] = p[3] = 0;    /* fsuid, suid, euid, and uid are initialized as 0 (root ID). */
    p[4] = p[5] = p[6] = p[7] = 0;    /* fsgid, sgid, egid, and gid are initialized as 0 (root ID). */
    p = (uint *) ((char *)(p + 8) + sizeof(void *));
    p[0] = p[1] = p[2] = ~0;          /* cap_permitted, cap_inheritable, and cap_effective */
    break;                            /* are initialized as 0xffffffff (full capability).  */
   }
   p++;
 }

 exit_kernel();
}
```

Figure 8. The attack code of the vmsplice root exploit. The comments are not part of the exploit code.

from 2.6.17 to 2.6.24.1, therefore many Linux distributions were vulnerable including one of famous distributions, Fedora Core 6, that is tested here.

Tracing Policies: In order to analyze this attack we use simple tracing policies. If user code runs in kernel mode, its execution is traced; If a memory access occurs, the accessed address and the value are traced as well. Typically the execution of user code in kernel mode should be prohibited because a potentially malicious user code can subvert the entire system by overwriting kernel code or data. In this attack example, it occurs by exploiting vulnerable kernel code.

Identifying Kernel Memory Victims: In this section, we show the effectiveness of LiveDM by identifying the attack victims of the vmsplice root exploit. We run a widely available exploit code from [23], and Table 3 summarizes the manipulated kernel memory. Note that we do not assume any knowledge about the attack, instead we only use the memory accesses captured by the above policies.

Let us illustrate how LiveDM identifies the first kernel memory victim shown in the first row of Table 3. When user code overwrites kernel memory, LiveDM searches the kernel memory map for the overwritten address and retrieves the runtime identifier (type ID) of the matched dynamic memory block. This type ID is as shown in column **Type ID**, and the field ID is determined as the offset of the address in the block range (shown in column **Field ID**). This pair of identifiers is stored in the trace and further translated into a data type and a field name offline. Using debugging information, the type ID is converted to a source code position, `kernel/fork.c:164`. In the code, the address of the allocated memory is assigned to the pointer variable on the left hand side of the statement. This variable has the pointer type of `task_struct` as shown in column **Data Type**. This is the type that the allocated memory block has. Using its data type definition, the field ID (`0x158`) is converted to a field, `fsuid`

Rootkit Name	T	Runtime Identification		Offline Interpretation	
		Type ID	**Field ID**	**Type** (D) / **Module Object** (M)	**Field / Offset**
adore-ng 0.53	D	fs/proc/generic.c:436	0x20	proc_dir_entry	get_info
(adore-ng.c	D	kernel/fork.c:610	0x4,12c,130	task_struct	flags,uid,euid
for Linux 2.4)	D	kernel/fork.c:610	0x134,138,13c	task_struct	suid,fsuid,gid
	D	kernel/fork.c:610	0x140,144,148	task_struct	egid,sgid,fsgid
	D	kernel/fork.c:610	0x1d0	task_struct	cap_effective
	D	kernel/fork.c:610	0x1d4,1d8	task_struct	cap_inheritable,cap_permitted
	M	-	-	ext3:ext3_dir_operations	readdir
	M	-	-	ext3:ext3_file_operations	write
knark 0.59	D	fs/proc/generic.c:436	0x38	proc_dir_entry	read_proc
	D	kernel/fork.c:610	0x4	task_struct	flags
kbdv3	D	kernel/fork.c:610	0x12c,130	task_struct	uid,euid
	D	kernel/fork.c:610	0x13c,140	task_struct	gid,egid
adore 0.42	D	fs/namespace.c:44	0x28	vfsmount	mnt_count
	D	fs/dcache.c:619	0x0	dentry	d_count
	D	kernel/fork.c:610	0x4,12c,130	task_struct	flags,uid,euid
	D	kernel/fork.c:610	0x134,138,13c	task_struct	suid,fsuid,gid
	D	kernel/fork.c:610	0x140,144,148	task_struct	egid,sgid,fsgid
	D	kernel/fork.c:610	0x1d0	task_struct	cap_effective
	D	kernel/fork.c:610	0x1d4,1d8	task_struct	cap_inheritable,cap_permitted
linuxfu	D	kernel/fork.c:610	0x50,54	task_struct	next_task,prev_task
hp 1.0.0	D	kernel/fork.c:610	0x50,54,9c	task_struct	next_task,prev_task,p_ysptr
	D	kernel/fork.c:610	0x98,a0,ac	task_struct	p_cptr,p_osptr,pidhash_next
	D	kernel/fork.c:610	0xb0,78	task_struct	pidhash_pprev,pid
SucKIT 1.3a	D	kernel/fork.c:610	0x4,c	task_struct	flags,addr_limit
superkit	D	kernel/fork.c:610	0x4,c	task_struct	flags,addr_limit
adore-ng 0.53	D	kernel/fork.c:248	0xc,1d0,1d4	task_struct	flags,uid,euid
(adore-ng-2.6.c	D	kernel/fork.c:248	0x1d8,1dc,1e0	task_struct	suid,fsuid,gid
for Linux 2.6)	D	kernel/fork.c:248	0x1e4,1ec,1f4	task_struct	eguid,fsguid,cap_effective
	D	kernel/fork.c:248	0x1f8,1fc	task_struct	cap_inheritable,cap_permitted
	M	-	-	ext3:ext3_dir_operations	readdir
	M	-	-	ext3:ext3_file_operations	write
	M	-	-	unix:unix_dgram_ops	rcvmsg
	M	-	-	ipv6:__this_module	offset 0x8

Table 4. Dynamic objects manipulated by rootkits. T: The kind of memory (D: dynamic and M: module). (OS: Redhat 8 for Linux 2.4 rootkits and Debian Sarge for Linux 2.6 rootkits).

(shown in column **Field**). The rest victims in Table 3 are interpreted in the same way. From this result, we can understand that this exploit aims at obtaining the root privilege by manipulating the user credential information in the kernel memory.

In order to confirm the correctness of this result, a snippet of the attack code is presented in Figure 8. This code first obtains the address of current process' PCB by calling the `get_current` function. Then it scans memory to identify specific fields for user credentials using a pattern of values. Once they are found, this function overwrites IDs with `0x0` (root ID) and capabilities with $\sim$`0` (=`0xffffffff`, full capability) to make the attacker a root user. This is exactly matched with the result presented in Table 3.

5.2 Dynamic Kernel Objects Manipulated by Real World Kernel Rootkits

Kernel rootkits target dynamic objects to conceal their activities because the locations and types of such objects are comparatively difficult to identify compared to static objects whose information is available at compile time. In this section LiveDM exposes the types of dynamic kernel objects manipulated by real world kernel rootkits. LiveDM can perform accurate translation of dynamically changing kernel memory because the kernel memory map is accurately updated with memory allocation/deallocation events in the guest machine.

Tracing Policies: LiveDM recognizes rootkit activity using previously proposed techniques for kernel rootkit detection [20] and prevention [24, 25]. Any execution of rootkit code is traced along with the memory access

Rootkit Name	Kernel Memory Function Call			Allocated Memory	
	Call Type	Caller	Extracted Caller Name	Size (Bytes)	Use
Rial	Allocation	Rial:0x17f	new_open:0x5b	14	Data: string
	Allocation	Rial:0x2e6,2fe	new_read:0x56,6e	25	Data: string
	Free	Rial:0x234	new_open:0x110	-	-
	Free	Rial:0x73c,74a	new_read:0x4ac,4ba	-	-
knark 0.59	Allocation	knark:0x1341,1371,1396	init_module:0x2d,5d,82	8,12,20	Data: rootkit data, string
	Free	knark:0x16af,16c0,16dc	cleanup_module:0xf7,108,124	-	-
	Free†	knark:0xe1a	knark_execve:0x62	-	-
kbdv3	Allocation	kbdv3:0x8a	bd_utime:0x2a	256	Data: string
adore 0.42	Allocation	adore:0x568	n_getdents64:0x8c	704	Data: rootkit data
	Allocation	adore:0xaa6	fp_get:0x4a	4096	Data: string
	Free	adore:0x5d8	n_getdents64:0xfc	-	-
	Free	adore:0xa55	fp_put:0x2d	-	-
	Free†	adore:0xaf9,b78	fp_get:0x9d,11c	-	-
SucKIT 1.3a	Allocation	Kernel:0xc010910f	system_call:0x33	13044	Code: rootkit installation
superkit	Allocation	Kernel:0xc010910f	system_call:0x33	12735	Code: rootkit installation
Synapsys-0.4	Allocation	Synapsys:0x79	hack_open:0x19	256	Data: string
	Allocation	Synapsys:0x872,8da	hack_write:0x156,1be	2000	Data: string
	Free	Synapsys:0x16b	hack_open:0x10b	-	-
	Free	Synapsys:0x8c1,92d	hack_write:0x1a5,211	-	-
override	Allocation	override:0x357,368	my_getdents64:0x42,53	64∼1024	Data: rootkit data, string
	Free	override:0x414,41d	my_getdents64:0xff,108	-	-
phalanx-p6	Allocation	Kernel:0xc0124375	sys_setdomainname:0x1e	4096	Data: buffer

Table 5. Dynamic memory allocation and free by rootkits. Kernel memory functions: kmalloc (Allocation), kfree (Free), kmem_cache_free (Free†). (OS: Redhat 8 for Linux 2.4 rootkits and Debian Sarge for Linux 2.6 rootkits).

targets. If invariant system components (e.g., kernel code, system call table, and interrupt descriptor table) are manipulated, such events are traced as well.

Identifying Kernel Memory Victims: Table 4 presents the list of dynamic kernel memory victims manipulated by real world kernel rootkits. We experimented with 13 rootkits in Linux 2.4 and 2.6 kernels, and 9 rootkits exhibited the behavior that manipulates dynamic or module objects. Most of these rootkits target static kernel data as well, however they are not presented because the translation of their addresses is straightforward by using the kernel symbol table (i.e., System.map in Linux). We focus on the cases of dynamic kernel objects whose addresses are dynamically determined during the execution of the guest OS.

LiveDM uses the kernel memory map that contains the address ranges of dynamic objects and their IDs. Thus, it can instantly retrieve the runtime ID of the accessed object (shown in column **Runtime Identification**) by searching the map with the address. In the offline process, the type ID and field ID pair are interpreted into a data type and a field name, and the result is presented in column **Offline Interpretation**. Module objects are interpreted by mapping their offsets within the module memory to the module symbols, which are extracted immediately after the loading of the module. Based on these victims, we can understand the system components to which the rootkits aim during attacks and the potential impact to users. The details of this kind of interpretation are explored in a related work [4].

5.3 Dynamic Rootkit Memory

The dynamic kernel objects identified in the previous section are allocated by legitimate kernel code. However, it is not the only code that can allocate or free dynamic objects. Rootkits can also instantiate dynamic objects. We analyzed 13 real world rootkits, 9 of which use own dynamic kernel memory. Identifying this memory is difficult because it has a dynamically determined address out of the loaded rootkit memory. In addition, such memory does not exhibit different characteristics from legitimate kernel memory just based on their content or addresses; therefore it is challenging to differentiate rootkit memory just using kernel memory layout. Dynamic

rootkit memory has not been previously well addressed in related approaches [24, 4]; however, this memory is important to understanding rootkit behavior because it serves as stealthy runtime storage that can contain code or data.

Tracing Policies: LiveDM uses the policies of the previous section to recognize rootkit activities. If a memory allocation function is called from the rootkit code, the allocated memory is considered to be rootkit memory. In addition, any anomaly in the memory allocation or deallocation call site is checked. If any code that is not used for allocation in normal execution is used during rootkit experiments to allocate kernel memory, it is traced and inspected. This execution patten is observed from rootkits that call a kernel memory allocation function using a system call indirectly.

Capturing Dynamic Rootkit Memory: Table 5 presents a summary of the dynamic kernel memory allocation and deallocation performed by 9 real world kernel rootkits. The information about the invoked dynamic kernel memory functions is presented in column **Kernel Memory Function Call**. For example, the first row in Table 5 shows that the code at the offset 383th bytes (i.e., `0x17f`) of the RIAL rootkit code allocates 14 bytes of dynamic memory by calling the `kmalloc` function. This code is placed in the 91st (i.e., `0x5b`) byte in the `new_open` rootkit function as shown in column **Extracted Caller Name**. We extract these symbols by parsing the rootkit binary of the ELF format when it is loaded. These names are required to exist in the binary to load the rootkit into the kernel (for the code relocation), but they have no relevance other than as the function names that rootkit authors use in the code.

An interesting result is shown by a group of rootkits that manipulate kernels from user space: SucKIT, superkit, and phalanx-p6. SucKIT and superkit replace one of the system call table entries with the code address of `kmalloc`, then call the `kmalloc` function through a system call. This call is invoked by an unmodified call instruction in the `system_call` function. However, this activity is still captured because the call site of `kmalloc` used by the rootkits is not part of the `kmalloc` callers in the kernel binary. Interestingly enough, the call instruction is indeed modified later (not shown in Table 4 since kernel code is a static target), but the modification occurs after the memory allocation. In contrast, phalanx-p6's `kmalloc` call is invoked by the injected code in the `sys_setdomainname` function and is therefore easily captured. This behavior previously could not be analyzed in PoKeR [4] because the executed memory activities are part of legitimate kernel code execution, which PoKeR does not trace to understand rootkit behavior. However, LiveDM provides a unique opportunity to observe this new aspect of rootkit behavior since it captures memory allocation activities for the entire kernel execution.

Following is the analysis of how the allocated memory is used by rootkits. This information is presented in column **Allocated Memory**. We classify the use of this memory depending on whether it is used for code execution or data. SucKIT and superkit load the allocated memory with the hooking code and execute it. Most rootkits use the memory to store data. Since we cannot assume the availability of source code for rootkits, our type derivation method is not applicable to this memory. Instead we use derivatives of the approaches [26, 11] that infer the layout of memory. If we obtain a layout, we mark the memory as rootkit data. However, in many cases the memory is used as an array involved with string functions. In such cases, we consider the memory as being used as a string. We confirm our inference is matched with the actual use by manual inspection of rootkit code.

In general, dynamic kernel memory is a useful resource for rootkits due to its flexible handling of storage. The knark rootkit uses dynamic memory to create its own `proc` device objects installed in the kernel, which provide a convenient backdoor mechanism to the attacker. More importantly, the use of this memory for code enables a sophisticated attack vector to bootstrap kernel mode execution without using a conventional LKM mechanism [27]. For example, SucKIT and superkit use dynamic memory for this purpose. These rootkits are user programs that do not have kernel memory space available, unlike LKM-based rootkits. Therefore they use dynamic kernel memory to place and execute the rootkit installation code in the kernel mode. By tracking the dynamic memory allocated by kernel rootkits, LiveDM can conveniently uncover such complicated behaviors.

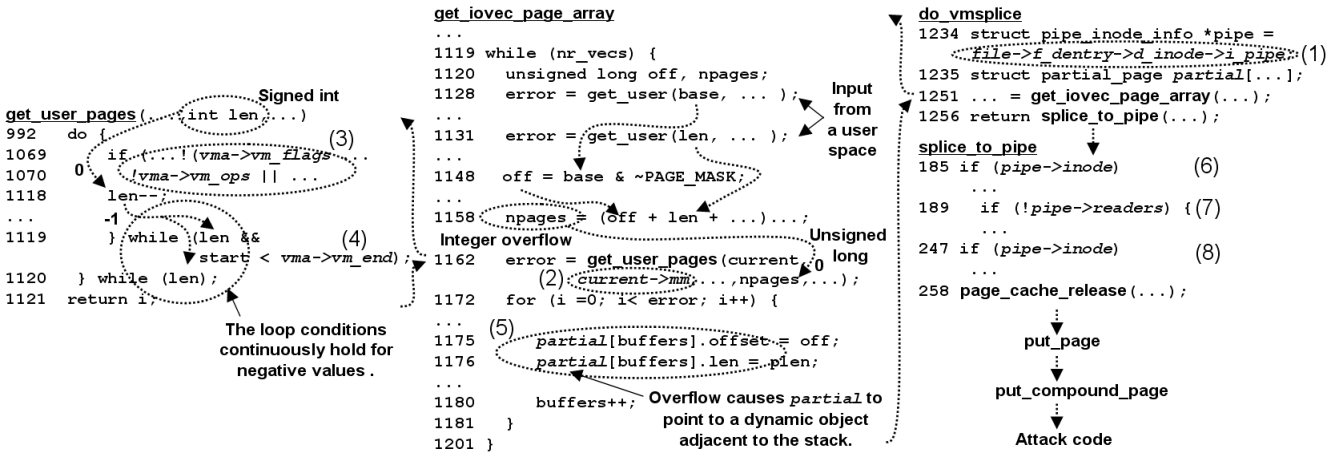


Figure 9. The control flow of vmsplice exploit attack: from do_vmsplice to the Attack code going through get_iovec_page_array, get_user_pages, and splice_to_pipe. (OS: Fedora Core 6).

5.4 Interpreting Dynamic Memory Targets in Kernel Execution

In this section, we will present a microscopic kernel debugging scenario assisted with LiveDM's dynamic memory interpretation. As we diagnose kernel execution over a traced period, LiveDM reveals the dynamic data structures that have been dereferenced at runtime for given code, if any. Therefore, kernel developers can validate whether each code accesses the correct object meant in the code semantic. An unusual memory object that is dereferenced for any reason, such as an overflow, can be confirmed since its type will be identified by LiveDM. Note that the dynamic memory status can be inaccurate at any moment other than the time of the access; thus, snapshot-based approaches [12, 11] are required to make a snapshot at each access to achieve similar accuracy. LiveDM effectively extends the coverage of accurate dynamic kernel memory analysis from a snapshot to a period of time.

Following is an example that shows how LiveDM can assist a general kernel debugging procedure. This scenario diagnoses the vulnerable kernel code that allowed a vmsplice root exploit attack (CVE-2008-0009, CVE-2008-0010, and CVE-2008-0600). This attack is launched by a user program that has a limited privilege; however, it turns the current user into a super-user by triggering kernel bugs and overwriting the user's credentials stored in the kernel memory.

Tracing Policies: In order to analyze this case, we set the tracing scope as the list of kernel functions in the kernel call stack when user-level exploit code manipulates kernel data in the kernel mode. The traced kernel functions include `put_compound_page`, `sys_vmsplice`, `splice_to_pipe`, `get_user_page`, and `put_page`. Static functions such as `do_vmsplice` and `get_iovec_page_array` appear as part of `sys_vmsplice`.

Checking Memory Dereferences with Dynamic Memory Interpretation: We track down the problem by starting from the attack code, then browsing the execution in reverse chronological order. An example of the attack code sequence is denoted with dotted arrows in Figure 9 and the dynamic data types accessed by this code are shown in Table 6. In order to highlight the unique assistance of LiveDM, we placed numbers in parentheses in Figure 9, Table 6, and the description.

Before the attack code is launched, `put_compound_page` is the last kernel function executed. This function deallocates memory using a given custom deallocator. The attack code is executed as this deallocator, which explains why untrusted user-level code is permitted to run in the kernel mode. Delivery of the attack code to this function is the art of this attack and it is enabled by the vulnerable code in the vmsplice system call.

In `splice_to_pipe`, one of the functions of which the vmsplice call is composed, there are three kinds of read

Case Number	Traced Code			Accessed Data Type on Normal Workload	Accessed Data Type on Exploit Execution
	Function	File	Line		
(1)	do_vmsplice	fs/splice.c	1234	file	file
(1)	do_vmsplice	fs/splice.c	1234	dentry	dentry
(1)	do_vmsplice	fs/splice.c	1234	inode	inode
(2)	get_iovec_page_array	fs/splice.c	1162	task_struct	task_struct
(3), (4)	get_user_pages	mm/memory.c	1069-1070, 1119	vm_area_struct	vm_area_struct
(5)	get_iovec_page_array	fs/splice.c	1175-1176	**thread_info**‡	**page***
(6), (7), (8)	splice_to_pipe	fs/splice.c	185, 189, 247	pipe_inode_info	pipe_inode_info

Table 6. Data types of dereferenced memory by the kernel code in Figure 9. Most code use the kernel stack (type: thread_info), but they are omitted except the case marked with ‡. The code in case (5) accesses an anomalous data type (marked with *) on exploit execution. (OS: Fedora Core 6).

accesses on dynamic objects other than the kernel stack. Such objects commonly are of the `pipe_inode_info` type and their field IDs are respectively translated into the fields (6) `inodes`, (7) `readers`, and (8) `inodes`. This result is exactly matched with the code at the file `fs/splice.c` in the lines 185, 189, and 247. The trace shows a specific execution path that the `if` statement at the line 189 has taken. The read value at this line is 0, and it confirms this control flow since the if condition (`!pipe->readers`) is satisfied with it.

LiveDM correctly identifies dynamic objects dereferenced in the program. For example, the code at the line 1234 of the `do_vmsplice` function has the expression comprised of three consecutive pointer dereferences. As we see in Table 6, (1) LiveDM accurately identifies three different dynamic objects described in the code.

In the trace of `get_iovec_page_array`, we found an excessive loop count returned from the `get_user_pages` function call. We could confirm this number of loops by counting the number of accesses on the dynamic kernel objects inside the loop. Among several dynamic objects from the trace, an object is repeatedly accessed along with the loop code. (4) LiveDM identifies its type and field are respectively `vm_area_struct` and `vm_end`, and this is matched with the object in the loop condition.

The loop count is consistently around 48 when the attack is successful. We determined it was enabled by a combination of several kernel bugs. In `get_iovec_page_array`, an unsigned long variable, `npages`, is set to 0 due to an integer overflow. This number is passed to a loop variable, `len`, in the `get_user_pages` function. This variable of a (signed) `int` type is expected to have a positive value and it is decreased in the do-while loop prior to the check of the loop condition ($len \neq 0$). However, due to an early decrement of the overflowed value (0), the loop conditions continuously hold for negative values, thus an excessive number of loops occur.

This abnormal loop count directly influences the access of the following variable; a local array `partial` placed in the stack is initialized in the loop at `fs/splice.c:1172-1181`. Due to the excessive loop count, this loop code overwrites the memory placed beyond this array at runtime. In our experiments the manipulation went beyond the current stack; therefore (5) the code overwrites the object that happens to be adjacent to the stack. This example highlights the capability of LiveDM, which can also identify the targets of wild memory accesses.

After the analysis, we determined that sanity checks were necessary for the values passed from user space. First, one of the `get_iovec_page_array`'s arguments `iov` is directly passed from the vmsplice system call and used to fill local variables `base` and `len` so it should be checked. Second, `base` and `len` should be carefully inspected as well after they are filled with the values of user space because they are the direct inputs that caused the overflow of `npages`. We confirmed that such vulnerabilities are patched in the later kernel versions.

6 Discussion

Since LiveDM operates in the VMM beneath the hardware interface, kernel malware cannot directly access LiveDM code or data. However, it can exhibit potential 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 and not use the legitimate kernel memory allocators that LiveDM observes. This behavior can be detected based on the intuition that memory allocators themselves use internal kernel data to maintain free and active memory chunks. LiveDM can check which kernel functions access such core memory structures. If LiveDM identifies unusual code other than the regular memory allocators that manipulate such data, it can be concluded that a custom memory allocator is present.

Another case is that malware manipulates a regular kernel control flow as in return-to-kernel/libc attacks [28, 29]. For example, malware can jump into the body of a memory allocator without passing the function entry. One way to determine this attack is to check the control flow integrity [30] of the kernel execution, which ensures that function calls occur to the function entries, thereby not allowing this kind of attack.

It may seem that a similar result to LiveDM can be achieved by setting breakpoints in a kernel debugger and manually browsing code. This process in fact describes part of what LiveDM accomplishes in an automated method, but it is not sufficient to achieve the goal that we target. The missed point here is that while such method manually obtains a mapping from a given code statement (breakpoint) to a type, there may be tens of thousands of runtime memory instances allocated from the code. Moreover, when malware targets one of a vast number of memory instances allocated by many different code statements, we need to connect the memory to its allocation code to identify the memory's type. LiveDM provides this critical mapping from a memory instance to a code statement in addition to the mapping from a code statement to a type in order to translate a given kernel address into a data type completely.

7 Related Work

PoKeR [4] is a kernel rootkit profiler that analyzes multiple aspects of kernel rootkit behavior. PoKeR can identify the attack targets of kernel rootkits on static and dynamic kernel memory. Unlike LiveDM, it assumes a rootkit behavior (i.e., it starts to access static objects first, then progressively follows pointers to find the targets). We demonstrated in Section 4.3 that this assumption can be violated by several attack techniques and rootkits can thus elude this profiler. In addition, since it relies on the rootkit activity to determine types, its analysis focus is limited to the memory manipulated by rootkits. In contrast, LiveDM can be used to analyze dynamic kernel memory accessed by legitimate kernel code as well as rootkit code without such limitations.

K-Tracer [31] can analyze the malicious behaviors of kernel rootkits in sensitive events using dynamic slicing techniques. Its algorithm requires determination of the sensitive data so it can be difficult to analyze DKOM attacks [2, 5] whose targets may not be predetermined. LiveDM, on the other hand, can derive the type of a dynamic object for a given address and is therefore applicable to a wider scope of problems, such as the analysis of legitimate kernel code execution as well as the interpretation of DKOM attack victims.

Several approaches have been proposed to infer data structures in a memory snapshot based on the memory analysis and static analysis. Laika [11] used Bayesian unsupervised learning to infer the layouts of data structures. KOP [12] improved the inference quality and achieved advanced recognition of generic pointers, type ambiguities, and arrays using the static analysis technique in addition to memory analysis. While these approaches use pointer values in the memory to construct a memory graph and map objects to types, LiveDM uses the allocation events to recognize kernel objects. So LiveDM is more tolerant to invalid pointer addresses or memory casting to generic types. In addition, when diagnosing a trace of dynamic kernel execution, it will be a challenge to reflect dynamically changing memory status for snapshot-based approaches unless they generate a memory graph for each dynamic memory change.

WIT [32] is an inline reference monitor that can check the validity of memory accesses. WIT assigns colors to newly allocated memory objects using memory wrapper functions. These colors and type IDs both serve as the names of dynamic objects. WIT uses static analysis to determine these names, and instrumentation is necessary to manage them in the system. In LiveDM, names (i.e., type IDs) are systematically extracted using standard rules about runtime context called function call conventions, so no change is necessary inside the guest system. Another difference is that WIT targets user memory while LiveDM targets kernel memory.

To recognize the activity of rootkits, LiveDM relies on previously proposed approaches in kernel rootkit defense [20, 24, 25]. We developed techniques to capture an individual dynamic kernel object along with runtime information to derive its type, which can be described as a fine-grained method of virtual machine introspection, originally introduced by Livewire [33].

8 Conclusion

In this paper, we presented a memory interpretation system that can automatically translate dynamic kernel memory addresses into data types. LiveDM can analyze the dynamic kernel objects accessed by the entire kernel execution because LiveDM identifies the types of the dynamic objects by using their memory allocation code. This approach significantly expands the analysis coverage, which was previously limited to the targets of rootkits with assumptions on attack mechanisms [4]. Also it enables an accurate analysis of volatile dynamic kernel memory over a continuous period of time, which previously was not possible by snapshot-based approaches [12, 11]. LiveDM is based on two novel techniques: (1) systematic identification of an individual dynamic object with the allocation code address (a type ID) and (2) static code analysis that automatically converts a type ID to a data type. Our prototype supports three off-the-shelf Linux distributions and we show LiveDM's general applicability and effectiveness in extensive case studies analyzing kernel malware and kernel bugs.

References

[1] N. L. Petroni and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[2] J. Butler, "DKOM (Direct Kernel Object Manipulation)," `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`.

[3] G. Hoglund, "Kernel Object Hooking Rootkits (KOH Rootkits).," `http://www.rootkit.com/newsread.php?newsid=501`.

[4] R. Riley, X. Jiang, and D. Xu, "Multi-Aspect Profiling of Kernel Rootkit Behavior," in *Proceedings of the 4th European Conference on Computer Systems (Eurosys'09)*, April 2009.

[5] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring," in *International Conference on Availability, Reliability and Security (ARES'09)*, 2009.

[6] MITRE Corporation, "Common Vulnerabilities and Exposures," `http://cve.mitre.org`.

[7] US-CERT, "US-CERT Vulnerability Notes Database," `http://www.kb.cert.org/vuls`.

[8] "The Month of Kernel Bugs (MoKB) archive." `http://projects.info-pull.com/mokb`.

[9] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'02)*, 2002.

[10] D. Evans, "Static Detection of Dynamic Memory Errors," *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 44–53, 1996.

[11] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging For Data Structures," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[12] 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'09)*, 2009.

[13] Free Software Foundation, "GCC, the GNU Compiler Collection," `http://gcc.gnu.org/`.

[14] V. Goyal, E. W. Biederman, and H. Nellitheertha, "Kdump, A Kexec-based Kernel Crash Dumping Mechanism," in *Proceedings of The Linux Symposium 2005 (OLS 2005)*. `http://lse.sourceforge.net/kdump`.

[15] VMware, Inc., "VMware Virtual Machine Technology.," `http://www.vmware.com`.

[16] Sun Microsystems, Inc, "VirtualBox," `http://www.virtualbox.org`.

[17] Parallels, "Parallels," `http://www.parallels.com`.

[18] F. Bellard, "QEMU: A Fast and Portable Dynamic Translator.," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005. `http://www.qemu.org`.

[19] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," in *Proceedings of 2008 USENIX Annual Technical Conference (USENIX'08)*, 2008.

[20] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings for the 13th USENIX Security Symposium*, August 2004.

[21] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh, "An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data," in *Proceedings for the 15th USENIX Security Symposium*, (Vancouver, B.C., Canada), July 2006.

[22] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory," in *Digital Investigation Journal 3(4):197-210*, 2006.

[23] qaaz, "Linux Kernel 2.6.17 - 2.6.24.1 vmsplice Local Root Exploit," `http://milw0rm.com/exploits/5092`.

[24] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing," in *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[25] 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 21st Symposium on Operating Systems Principles (SOSP'07)*, ACM, 2007.

[26] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[27] K. J. Jones, "Loadable Kernel Modules," *;login: The Magazine of USENIX and SAGE, 26(7)*, November 2001.

[28] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings for the 18th USENIX Security Symposium*, 2009.

[29] H. Shacham, "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'07)*, (New York, NY, USA), pp. 552–561, ACM, 2007.

[30] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, 2005.

[31] A. Lanzi, M. Sharif, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, 2009.

[32] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *Proceedings of IEEE Symposium on Security and Privacy, 2008 (SP 2008)*, May 2008.

[33] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.