Kernel Malware Analysis with Untampered and Temporal Views of Dynamic Kernel Memory

*Junghwan Rhee, *Ryan Riley, *Dongyan Xu, **Xuxian Jiang

* Department of Computer Science, Purdue University
* Department of Computer Science and Engineering, Qatar University
** Department of Computer Science, North Carolina State University
Outline

- Background
- Allocation-driven mapping
- Evaluation
- Discussion
- Conclusion
- Demo
Kernel malware

- Kernel malware attacks operating system kernels.
  - e.g., kernel rootkits

- Attack goals
  - Hide processes, files, etc.
  - Provide hidden services, backdoors, etc.

- Attack techniques
  - Hijack system services (e.g., system calls)
  - Directly manipulate kernel data (DKOM)
  - Hijack hooks by overwriting function pointers (KOH)
Kernel malware

- Kernel malware attacks operating system kernels.
  - e.g., kernel rootkits

- Attack goals
  - Hide processes, files, etc.
  - Provide hidden services, backdoors, etc.

- Attack techniques
  - Hijack system services (e.g., system calls)
  - Directly manipulate kernel data (DKOM)
  - Hijack hooks by overwriting function pointers (KOH)
Kernel memory mapping has been used for kernel integrity checking and kernel malware detection.

Existing approaches

**Type-projection mapping**: kernel objects identification by recursively traversing pointers from global objects

- Static: memory snapshots as input
- Dynamic: memory traces as input
Related work

- Type-projection mapping using memory snapshots
  - SBCFI [CCS 2007]
  - Gibraltar [ACSAC 2008]
  - KOP [CCS 2009]

- Type-projection mapping using memory traces
  - Rkprofiler [RAID 2009]
  - PoKeR [Eurosys 2009]
**Type-projection mapping**

Data type definition of X:

```c
struct X {
    struct X *next;
    struct X *prev;
}
```

A memory snapshot:

**Static memory**

```
S_1
```

- X *next
- X *prev

**Dynamic memory**
Type-projection mapping

Data type definition of X

```
struct X {
    struct X *next;
    struct X *prev;
}
```

A memory snapshot

Static memory

- Address
- Value
- X *next
- X *prev

Dynamic memory

- Address
- Value
- X *next
- X *prev

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory
The map of kernel objects is subject to the manipulation by malware.
X1, X2, and X3 : kernel objects allocated in the \textit{same address} with the \textit{same data type}.

- A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.
X1, X2, and X3: kernel objects allocated in the same address with the same data type.

A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.
X1, X2, and X3: kernel objects allocated in the same address with the same data type.

A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.
Kernel objects are identified by transparently capturing kernel memory function calls.

The memory ranges are extracted from function arguments and return values.

Call stack information (allocation call site) is used to derive data types.

* An memory allocation call site: code address of a memory allocation call.
Allocation-driven mapping

Lifetime of a dynamic kernel object

Allocation  Usage  Deactivation
Advantages

- Un-tampered view
  - Tolerant to the manipulation of memory content
Advantages

- Un-tampered view
  - Tolerant to the manipulation of memory content
- Temporal view
  - Lifetime of dynamic data is tracked to differentiate objects at the same memory location
Techniques: Map generation

Kernel source code

```
a = kmalloc(size, flag);
```

* An memory allocation call site: code address of a memory allocation call

[Diagram showing the relationship between Guest OS, VMM, Kernel memory, Registers, Kernel stack, Kernel object map, and a map entry for an object.]
Techniques: Map generation

**Kernel memory**

**Registers**

**Kernel stack**

**Kernel object map**

**Kernel source code**

```c
a = kmalloc (size, flag);
```

* An memory allocation call site: code address of a memory allocation call
Techniques: Map generation

```
a = kmalloc(size, flag);
```

* An memory allocation call site: code address of a memory allocation call
Techniques: Map generation

*An memory allocation call site: code address of a memory allocation call
Techniques: Type derivation

Kernel source code

Modified Compiler

Extracted code elements

Static analysis

Data types

Memory allocation call sites*

Debugging Information

Allocation code statements

A type definition

\[
T: \text{ struct } X \{
  \text{int } a*; \\
\};
\]

A declaration of a pointer

\[
D: \text{ struct } X \ast a;
\]

An assignment statement

\[
A: \text{ a } = \text{kmalloc (size, flag)};
\]

* An memory allocation call site: code address of a memory allocation call
Implementation

- LiveDM: Live Dynamic kernel memory Map
- Supported guest OS kernels
  - Redhat 8, Debian Sarge, Fedora Core 6
- Virtual machine monitor: QEMU
- Knowledge of kernel memory functions is assumed.
- Type resolution
  - Debugging symbols for translation of allocation call sites
  - Modified gcc compiler to extract code elements
Evaluation

- Effectiveness
- Performance
- Applications
  - Hidden object detector (un-tampered view)
  - Temporal malware behavior monitor (temporal view)
### Evaluation: Identifying Objects

#### Type Resolution

<table>
<thead>
<tr>
<th>Call Site</th>
<th>Declaration</th>
<th>Data Type</th>
<th>Case #Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel/fork.c:248</td>
<td>tsk</td>
<td>task_struct</td>
<td>66</td>
</tr>
<tr>
<td>kernel/fork.c:795</td>
<td>sig_hand_struct</td>
<td>sig_hand_struct</td>
<td>63</td>
</tr>
<tr>
<td>fs/exec.c:587</td>
<td>signal_struct</td>
<td>signal_struct</td>
<td>1</td>
</tr>
<tr>
<td>kernel/fork.c:526</td>
<td>mm_struct</td>
<td>mm_struct</td>
<td>7</td>
</tr>
<tr>
<td>kernel/fork.c:271</td>
<td>vm_area_struct</td>
<td>vm_area_struct</td>
<td>149</td>
</tr>
<tr>
<td>mm/mmap.c:748</td>
<td>vm_area_struct</td>
<td>vm_area_struct</td>
<td>1004</td>
</tr>
<tr>
<td>mm/mmap.c:1521</td>
<td>vm_area_struct</td>
<td>vm_area_struct</td>
<td>5</td>
</tr>
<tr>
<td>mm/mmap.c:1657</td>
<td>vm_area_struct</td>
<td>vm_area_struct</td>
<td>48</td>
</tr>
<tr>
<td>fs/exec.c:402</td>
<td>vm_area_struct</td>
<td>vm_area_struct</td>
<td>47</td>
</tr>
<tr>
<td>kernel/fork.c:677</td>
<td>file_struct</td>
<td>file</td>
<td>53</td>
</tr>
<tr>
<td>kernel/fork.c:597</td>
<td>buffer_head</td>
<td>buffer_head</td>
<td>828</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>bdev_inode</td>
<td>bdev_inode</td>
<td>5</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>dentry</td>
<td>dentry</td>
<td>4203</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>ino</td>
<td>ino</td>
<td>1209</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>vfsmount</td>
<td>vfsmount</td>
<td>16</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>proc_inode</td>
<td>proc_inode</td>
<td>237</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>request_queue_t</td>
<td>request_queue_t</td>
<td>18</td>
</tr>
<tr>
<td>fs/file_table.c:69</td>
<td>io_context</td>
<td>io_context</td>
<td>10</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>socket_alloc</td>
<td>socket_alloc</td>
<td>12</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>sock</td>
<td>sock</td>
<td>3</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>dentry</td>
<td>dentry</td>
<td>1</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>neighbour</td>
<td>neighbour</td>
<td>1</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>tcp_bind_bucket</td>
<td>tcp_bind_bucket</td>
<td>4</td>
</tr>
<tr>
<td>fs/block_dev.c:232</td>
<td>fib_node</td>
<td>fib_node</td>
<td>9</td>
</tr>
</tbody>
</table>

#### Identified Instances

```c
248 tsk = kmem_cache_alloc(...);
```

A list of core dynamic kernel objects (OS: Debian Sarge)

Total dynamic kernel objects: 29488
Evaluation: Identifying objects

<table>
<thead>
<tr>
<th>Call Site</th>
<th>Declaration</th>
<th>Data Type</th>
<th>Case #Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel/fork.c:248</td>
<td>task_struct *tsk;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kernel/fork.c:243</td>
<td>tsk = kmem_cache_alloc(...);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kernel/fork.c:248</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>kernel/fork.c:243</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>kernel/fork.c:248</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>kernel/fork.c:243</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488
A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488
Manual analysis: convert allocation call sites to data types (similar to validation methods of KOP [Carbone et. al., CCS 2009] and Laika [Cozzie et. al., OSDI 2008])
Evaluation: Performance

- **Benchmarks**
  - Kernel compile, UnixBench, nbench

- **Overhead**
  - Slowdown compared to unmodified QEMU (worst in benchmarks): 42% for Linux 2.4, 125% for Linux 2.6
  - Mainly caused by the capture of dynamic objects
  - Near-zero overhead for CPU-intensive benchmarks

- **Non-production application scenarios**
  - Honeypot, malware profiling, kernel debugging
Hidden object detector
  - Periodic comparison of an allocation-driven map and memory content
Hidden object detector

- Periodic comparison of an allocation-driven map and memory content
- **Hidden object detector**
  - Periodic comparison of an allocation-driven map and memory content
  - 10 kernel rootkits are tested and all detected.
  - Agnostic to the injection of malware code
  - Non-code injection attacks (hide_lkm and fuulld) are detected.

| Rootkit Name | $|L| - |S|$ | Manipulated Data | Operating System | Attack Vector |
|--------------|-----------------|-----------------|------------------|---------------|---------------|
| hide_lkm     | # of hidden modules | module next     | Redhat 8         | /dev/kmem     |
| fuulld       | # 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           |
| modhidel1    | # of hidden modules | module next     | Redhat 8         | LKM           |
| kis 0.9 (server) | # of hidden modules | module next | Redhat 8         | LKM           |
| adore-ng-2.6 | # of hidden modules | module list.next.list.prev | Debian Sarge     | LKM           |
| ENYELKM 1.1  | # of hidden modules | module list.next.list.prev | Debian Sarge     | LKM           |

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

- Systematically visualize malware influence via the manipulation of dynamic kernel memory
- Steps

![Diagram showing steps involving causes and effects, with a log of kernel control flow, memory accesses, and an allocation-driven map log over time.](image)
Temporal Malware Behavior Monitor

- Systematically visualize malware influence via the manipulation of dynamic kernel memory

<table>
<thead>
<tr>
<th>Call Site</th>
<th>Offset</th>
<th>Type / Object (Static, Module object)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork.c:610</td>
<td>0x4,12c,130</td>
<td>task_struct</td>
<td>flags,uid,euid</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>0x134,138,13c</td>
<td>task_struct</td>
<td>suid,fsuid,gid</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>0x140,144,148</td>
<td>task_struct</td>
<td>egid,sgid,fsgid</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>0x1d0</td>
<td>task_struct</td>
<td>cap_effective</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>0x1d4</td>
<td>task_struct</td>
<td>cap_inheritable</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>0x1d8</td>
<td>task_struct</td>
<td>cap_permitted</td>
</tr>
<tr>
<td>generic.c:436</td>
<td>0x20</td>
<td>proc_dir_entry</td>
<td>get_info</td>
</tr>
</tbody>
</table>

The list of kernel objects manipulated by adore-ng rootkit
An application of the temporal view

Kernel control flow

Memory accesses to T3’s address (+:read, x:write)

Time (Billions of instructions)
An application of the temporal view

Kernel control flow

Memory accesses to T3’s address (+:read, x :write)

Before attack

After attack

(Billions of instructions)

Time

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory
An application of the temporal view

Kernel control flow

Before attack

After attack

Memory accesses to T3’s address (+:read, x:write)

Allocation-driven map log

The time range relevant to the attack

(Teen Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory)
Malware analysis is guided to the attack victim objects (e.g., T₃).
Malware analysis using a data view

Before the rootkit attack

After the rootkit attack

Kernel object maps

- task_struct (PCB)
- proc_dir_entry
- kernel modules
- rootkit
- ext3

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

38
Malware analysis using a data view

Kernel object maps
- task_struct (PCB)
- proc_dir_entry
- kernel modules
- rootkit
- ext3

Before the rootkit attack
- PCB status:
  - uid = euid = 500
  - suid = fsuid = 500
  - gid = egid = 500
  - fsgid = 500
  - cap_effective = cap_inheritable = cap_permitted = 0
- User credentials:
  - uid = euid = 500
  - suid = fsuid = 500
  - gid = egid = 500
  - fsgid = 500
  - cap_effective = cap_inheritable = cap_permitted = 0

After the rootkit attack
- PCB status:
  - uid = euid = 0
  - suid = fsuid = 0
  - gid = egid = 0
  - fsgid = 0
  - cap_effective = cap_inheritable = cap_permitted = 0xffffffff
- Root credentials:
  - uid = euid = 0
  - suid = fsuid = 0
  - gid = egid = 0
  - fsgid = 0
  - cap_effective = cap_inheritable = cap_permitted = 0xffffffff

Privilege escalation attack
Before the rootkit attack

Kernel control flow graphs
Before the rootkit attack

Kernel control flow graphs

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory
Malware analysis using a code view

Before the rootkit attack

After the rootkit attack

Kernel control flow graphs
Malware analysis using a code view

Kernel control flow graphs

Before the rootkit attack

After the rootkit attack

Execution time

Kernel Code

Hijacked hook activity

Read of a function pointer

Redirection

Read of a function pointer

Kernel control flow graphs
Memory objects of 3rd party drivers, malware
  - Source code is required to derive data types.

Memory aliasing (type casting)
  - Allocation-driven map does not have aliasing problem by avoiding the evaluation of pointers.
  - Allocation using generic pointers: 0.1% of total objects

Attack cases towards memory functions
Un-tampered and temporal views of dynamic kernel objects can be enabled for malware analysis.

- Kernel data hiding attacks can be detected by using an un-tampered view.
- Temporal view can guide a malware analyzer to attack victim objects by tracking data lifetime.
Main technique: Live kernel object map
  - Live status is dumped to a GUI every 5 seconds.
  - Dynamic changes of the map are illustrated.

Applications: Hidden PCB and module detector
  - HP rootkit hides processes.
  - modhide rootkit hides kernel modules (drivers).
  - Data hiding attacks are checked every 5 seconds.

URL:

Note: some parts of a video clip are trimmed to reduce its play time.
Thank you, Questions?