Kernel Malware Analysis with Un-tampered 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

Static memory

Address

$S_1$

Value

X *next

X *prev

Dynamic memory

Address

Data type definition of X

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

A memory snapshot
Type-projection mapping

Data type definition of X

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

A memory snapshot
The map of kernel objects is subject to the manipulation by malware.
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.
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

Allocation  Usage  Deallocation

Lifetime of a dynamic kernel object
Advantages

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

Lifetime of a dynamic kernel object
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 memory

Registers

Kernel stack

Kernel object map

VMM

Guest OS

A map entry for an object

Kernel source code

Allocation

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

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

Kernel stack
Kernel memory
Registers

Kernel object map

VMM
{ a, a+size }, A map entry for an object

Guest OS

Kernel source code

Allocation

a = kmalloc (size, flag);

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

Kernel stack

Registers

Kernel memory

Kernel object map

Guest OS

Call site

Kernel source code

Allocation

a = kmalloc (size, flag);

VMM

A map entry for an object

Runtime identifier: a memory allocation call site*

* 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

Kernel memory

Registers

Kernel stack

Kernel object map

VMM

Guest OS

Kernel source code

Deallocation

kfree(a);
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: struct X {
    int a*;
};

A declaration of a pointer

D: struct X *a;

An assignment statement

A: a = 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)
The slide titled "Evaluation: Identifying objects" presents a list of core dynamic kernel objects for the Debian Sarge operating system. The slide includes a table with columns for call site, declaration, data type, and case number. For instance, the allocation statement in `kernel/fork.c:248` is shown as:

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

The total dynamic kernel objects are 29488.
A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488
Evaluation: Identifying objects

A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488

Type resolution

Identified instances

kernel/fork.c:248

Type definition

include/linux/sched.h

390 struct task_struct {
    ...
};

Declaration of a pointer

kernel/fork.c

243 struct task_struct *tsk;

Allocation statement

kernel/fork.c

248 tsk =
    kmem_cache_alloc(...);
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
An application of the un-tampered view

- Hidden object detector
  - Periodic comparison of an allocation-driven map and memory content
An application of the un-tampered view

- 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 fuuld) are detected.
Temporal Malware Behavior Monitor

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

- Steps

![Diagram showing temporal view of kernel control flow, memory accesses, and allocation map log over time.](image-url)
An application of the temporal view

- Temporal Malware Behavior Monitor
  - Systematically visualize malware influence via the manipulation of dynamic kernel memory

<table>
<thead>
<tr>
<th>Call Site</th>
<th>Type/Object</th>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>flags, uid, euid</td>
<td>0x4, 12c, 130</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>suid, fsuid, gid</td>
<td>0x134, 138, 13c</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>egid, sgid, fsgid</td>
<td>0x140, 144, 148</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>cap_effective</td>
<td>0x1d0</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>cap_inheritable</td>
<td>0x1d4</td>
</tr>
<tr>
<td>fork.c:610</td>
<td>task_struct</td>
<td>cap_permitted</td>
<td>0x1d8</td>
</tr>
<tr>
<td>generic.c:436</td>
<td>proc_dir_entry</td>
<td>get_info</td>
<td>0x20</td>
</tr>
<tr>
<td>C()</td>
<td>proc_root_inode_operations</td>
<td>lookup</td>
<td></td>
</tr>
<tr>
<td>C()</td>
<td>proc_root_operations</td>
<td>readdir</td>
<td></td>
</tr>
<tr>
<td>C()</td>
<td>unix_dgram_ops</td>
<td>recvmsg</td>
<td></td>
</tr>
<tr>
<td>(Module object)</td>
<td>ext3_dir_operations</td>
<td>readdir</td>
<td></td>
</tr>
<tr>
<td>(Module object)</td>
<td>ext3_file_operations</td>
<td>write</td>
<td></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)

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

Kernel control flow

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

Before attack

After attack

0.1 0.2 0.3 0.4 Time
(Billions of instructions)
An application of the temporal view

Kernel control flow

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

Allocation-driven map log

Before attack

After attack

T1’s lifetime
T2’s lifetime
T3’s lifetime
T4’s lifetime
T5’s lifetime

The time range relevant to the attack

(Billions of instructions)
Malware analysis is guided to the attack victim objects (e.g., T₃).
Malware analysis using a data view

Kernel object maps

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

Before the rootkit attack

After the rootkit attack
Malware analysis using a data view

Kernel memory address

PCB status
uid = euid = 500
suid = fsuid = 500
gid = egid = 500
fsgid = 500
cap_effective = cap_inheritable = cap_permitted = 0

User credentials

Before the rootkit attack

Root credentials

Privilege escalation attack

After the rootkit attack

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

Kernel control flow graphs
Before the rootkit attack

Kernel control flow graphs
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

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory
Memory objects of 3\textsuperscript{rd} party drivers, malware
\begin{itemize}
  \item Source code is required to derive data types.
\end{itemize}

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

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?