Virtual Machine Introspection

Bhushan Jain
Computer Science Department
Stony Brook University
Traditional Environment

Operating System
Traditional Environment

Operating System

Kernel Heap

Process Descriptors
Traditional Environment

Operating System

Kernel Heap

Process Descriptors
Traditional Environment

Operating System

Process Descriptors

Kernel Heap

Vulnerabilities
Traditional Environment

Process Descriptors

Kernel Heap

Operating System

Vulnerabilities
Traditional Environment

Privilege Escalation

Vulnerabilities

Process Descriptors

Kernel Heap

Operating System
Traditional Environment

Privilege Escalation

Vulnerabilities

Process Descriptors

Kernel Heap

Operating System
Traditional Environment

Privilege Escalation

Process Descriptors

Vulnerabilities

Kernel Heap

Operating System
Security monitors can be easily subverted by rootkits
Layered Security

- Guest OS
- Hypervisor

Guest Kernel Heap
Layered Security

Guest OS

Hypervisor

Guest Kernel Heap
Layered Security

Guest OS

Hypervisor

Guest Kernel Heap
Layered Security

Guest OS

Hypervisor

Guest Kernel Heap
Layered Security

Guest OS

Hypervisor

Narrower Interface

Guest Kernel Heap
Layered Security

Narrower Interface = Fewer exploitable vulnerabilities
Virtual Machine Introspection

Guest OS

Hypervisor

Guest Kernel Heap
Virtual Machine Introspection

- Guest OS
- Hypervisor
- Guest Kernel Heap
Virtual Machine Introspection

Guest OS

Hypervisor

Guest Kernel Heap

VMI
Virtual Machine Introspection

VMI: A technique to monitor the guest activities from VMM
Applications of VMI

- Introspect VM memory and CPU registers
- Introspect disk contents
- Network traffic
Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
- Network traffic
Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
  - E.g., Intrusion Prevention Systems
Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
  - E.g., Intrusion Prevention Systems

VMI is useful for more than just monitoring guest OS
VMI In Action

task_struct

pid_t pid

....

char comm[16]

task_struct
*next_task

task_struct
*prev_task

....
VMI In Action

```
struct task_struct {
    pid_t pid;
    ....
    char comm[16];
    struct task_struct *next_task;
    struct task_struct *prev_task;
    ....
};
```

Skype
VMI In Action
VMI In Action

init_task

Skype

init_task

Dropbox

init_task

```
task_struct
  pid_t pid
  ...
  char comm[16]
  task_struct *next_task
  task_struct *prev_task
  ...
```
VMI In Action

init_task

Init

Skype

Dropbox

\textbf{init_task}

\begin{itemize}
  \item \texttt{pid_t pid}
  \item char \texttt{comm[16]}
  \item task_struct *\texttt{next_task}
  \item task_struct *\texttt{prev_task}
\end{itemize}

\begin{itemize}
  \item \texttt{pid_t pid}
  \item char \texttt{comm[16]}
  \item task_struct *\texttt{next_task}
  \item task_struct *\texttt{prev_task}
\end{itemize}

\begin{itemize}
  \item \texttt{pid_t pid}
  \item char \texttt{comm[16]}
  \item task_struct *\texttt{next_task}
  \item task_struct *\texttt{prev_task}
\end{itemize}
VMI In Action

![Diagram of task_struct structures with pointers and variables]

init_task

Init

Skype

Dropbox

Typecast memory contents to structure definition
High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances

- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]

- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]
Learning and Reconstruction

Linux Guest OS

Hypervisor

Guest Kernel Heap

01010011
11110010
11010110
01101010
10111001
11010100
10011100
10101011
Learning and Reconstruction

Learning Phase: Generate data structure signature

Offline safe environment

Linux Guest OS

Hypervisor

Guest Kernel Heap

01010011 11110010 11010110 01101010
10111001 11010100 10011100 10101011

Stony Brook University
Learning and Reconstruction

Learning Phase: Generate data structure signature

Searching Phase: Find data structure instances

Offline safe environment

Continuous over guest lifespan
Learning and Reconstruction

**Learning Phase**: Generate data structure signature

**Searching Phase**: Find data structure instances

Assumption: Same OS behavior in learning and monitoring
Approaches for L&R

- **Learning**: Extract data structure signature
  - 
  - 
  - 

- **Searching**: Identify data structure instances
  - 
  - 
  -
Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures

- **Searching**: Identify data structure instances
Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis

- **Searching**: Identify data structure instances
Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning

- **Searching**: Identify data structure instances
  - 
  - 
  -
Approaches for L&R

**Learning**: Extract data structure signature
- Hand-crafted data structure signatures
- Source code analysis
- Dynamic Learning

**Searching**: Identify data structure instances
- Linearly scan kernel memory
-
Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning

- **Searching**: Identify data structure instances
  - Linearly scan kernel memory
  - Traverse data structure pointers
Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning

- **Searching**: Identify data structure instances
  - Linearly scan kernel memory
  - Traverse data structure pointers
  - Monitor object allocators
Learning Techniques

- Hand-crafted data structure signatures
- Source code analysis
- Dynamic Learning
Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]

- Source code analysis

- Dynamic Learning
Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]

- Source code analysis
  - Points-to analysis generates graph of kernel object types
  - Not all pointers in a data structure point to valid data
  - State of Art: MAS [5], SigGraph [6]

- Dynamic Learning
  
  - 
  
  -
Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]

- Source code analysis
  - Points-to analysis generates graph of kernel object types
  - Not all pointers in a data structure point to valid data
  - State of Art: MAS [5], SigGraph [6]

- Dynamic Learning
  - Supervised machine learning: train on a trusted OS
  - State of Art: RSFKDS [7]
Searching Techniques

- Linearly scan kernel memory

  Guest Kernel Heap

  01010011  10111001
  11110010  11010100
  11010110  10011100
  01101010  10101011

- Traverse data structure pointers
Searching Techniques

- Linearly scan kernel memory
  Guest Kernel Heap

- Traverse data structure pointers
Searching Techniques

- Linearly scan kernel memory
  
  Guest Kernel Heap

- Traverse data structure pointers

Chrome
- task_struct
  - pid_t pid
  - ....
  - char comm[16]
  - task_struct
    - *next_task
  - task_struct
    - *prev_task

Thunderbird
- task_struct
  - pid_t pid
  - ....
  - char comm[16]
  - task_struct
    - *next_task
  - task_struct
    - *prev_task

Firefox
- task_struct
  - pid_t pid
  - ....
  - char comm[16]
  - task_struct
    - *next_task
  - task_struct
    - *prev_task

Dropbox
- task_struct
  - pid_t pid
  - ....
  - char comm[16]
  - task_struct
    - *next_task
  - task_struct
    - *prev_task

Skype
- task_struct
  - pid_t pid
  - ....
  - char comm[16]
  - task_struct
    - *next_task
  - task_struct
    - *prev_task
But L&R is too involved

- L&R builds tools to mine information
  - This is hard!!!

- Can we just cheat?
  - Reuse the static guest kernel code
  - Make runtime kernel data and heap available to it

Just reuse the guest code to interpret kernel heap
High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances

- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]

- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]
Code Implanting

Linux Guest OS

Hypervisor

Guest Kernel Heap

01010011 11110010 11010110 01101010
10111001 11010100 10011100 10101011

Guest Kernel Heap
Code Implanting

```c
struct task_struct *task;
for_each_process(task)
    call_home("%s\n", task->comm);
return 0;
```
struct task_struct *task;
for_each_process(task)
    call_home("%s\n", task->comm);
return 0;
Code Implanting

Protect the injected code

struct task_struct *task;
for_each_process(task)
    call_home("%s\n",task->comm);
return 0;
Code Implanting

Protect the injected code

Inject code in guest OS; Difficult to protect

```c
struct task_struct *task;
for_each_process(task)
call_home("%s\n", task->comm);
return 0;
```
High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances

- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]

- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]
Data Outgrafting

Guest Kernel Heap

Linux Guest OS

COW

Guest Kernel Heap

01010011
11110010
11010110
01101010

01010011
11110010
11010110
01101010

01010011
11110010
11010110
01101010

01010011
11110010
11010110
01101010

Trusted Sibling Guest OS

Hypervisor
Data Outgrafting

Linux Guest OS

Guest Kernel Heap

01010011 11110010 11010110 01101010

01010011 11110010 11010110 01101010

01010011 11110010 11010110 01101010

01010011 11110010 11010110 01101010

Hypervisor

Guest Kernel Heap

01010011 11110010 11010110 01101010

01101010 10011100 10011100 10011100

01101010 10011100 10011100 10011100

01101010 10011100 10011100 10011100

Trusted Sibling Guest OS

Guest Kernel Heap

01010011 11110010 11010110 01101010

01101010 10011100 10011100 10011100

01101010 10011100 10011100 10011100

01101010 10011100 10011100 10011100

Hypervisor
Data Outgrafting

Reuse static trusted kernel code; Input runtime heap & data
# Listing all the running processes

| Userspace | Kernel |
Listing all the running processes

```
ps -a
```

Userspace

Kernel
Listing all the running processes

C-library call

```
readdir ("/proc")
```

ps -a

Userspace

Kernel
Listing all the running processes

C-library call

ps -a

Userspace

readdir ("/proc")

Kernel

sys_getdents ("/proc")

System call
Listing all the running processes

C-library call

Userspace

Kernel

sys_getdents ("/proc")

ps -a

System call

vfs_readdir ("/proc")

VFS call

readdir ("/proc")
Listing all the running processes

C-library call

Userspace

Kernel

readdir ("/proc")

sys_getdents ("/proc")

vfs_readdir ("/proc")

proc_root_file->f_op->readdir

System call

VFS call

FS specific call

ps -a
Listing all the running processes

C-library call

Userspace

Kernel

sys_getdents
("/proc")

System call

ps -a

vfs_readdir
("/proc")

VFS call

vfs_readdir
("/proc")

FS specific call

sys_getdents
("/proc")

proc_root_file->f_op->readdir

proc_root_file->f_op

*readdir()
Listing all the running processes

Readdir ("/proc")

sys_getdents ("/proc")

vfs.readdir ("/proc")

proc_root_file->f_op->readdir

proc_root_file->f_op

*readdir()

Return pids of all processes from list
## Trusted Guest Pervasive in VMI

<table>
<thead>
<tr>
<th>Technique</th>
<th>Approach</th>
<th>Trust Guest That</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn &amp; Reconstruct</td>
<td>Learn template then search</td>
<td>➢ Same OS behavior in learning and monitoring phases</td>
</tr>
<tr>
<td>Code Implanting</td>
<td>Monitoring inside guest OS</td>
<td>➢ Guest OS reports correct information</td>
</tr>
<tr>
<td>Data Outgrafting</td>
<td>Use sibling VM; share memory</td>
<td>➢ Identical guest OS behavior in monitored and trusted sibling VM</td>
</tr>
</tbody>
</table>
Semantic Gap

Guest OS

Hypervisor

Guest Kernel Heap
Semantic Gap

Guest OS

Hypervisor

Guest Kernel Heap

01010011  11110010  11010110  01101010
10111001  11010100  10011100  10101011

Guest Kernel Heap
Semantic Gap

Guest OS

Hypervisor

Guest Kernel Heap

Semantic Gap

Guest Kernel Heap

01010011
1110010
11010011
01101010
10111001
11010100
10011100
10101011
Semantic Gap
Semantic Gap

Need high level information; Available low level information
Semantic Gap: Details
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Variable values</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td></td>
<td>Objects and Types</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td></td>
<td>File system and Files</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td>Disk Data</td>
<td>File system and Files</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td>Disk Data</td>
<td>File system and Files</td>
</tr>
<tr>
<td></td>
<td>Interrupt/Exceptions</td>
</tr>
</tbody>
</table>
## Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td>Disk Data</td>
<td>File system and Files</td>
</tr>
<tr>
<td>Hardware Events</td>
<td>Interrupt/Exceptions</td>
</tr>
</tbody>
</table>
# Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td>Disk Data</td>
<td>File system and Files</td>
</tr>
<tr>
<td>Hardware Events</td>
<td>Interrupt/Exceptions</td>
</tr>
<tr>
<td></td>
<td>Packets, Buffers</td>
</tr>
</tbody>
</table>
# Semantic Gap: Details

<table>
<thead>
<tr>
<th>Hypervisor can observe</th>
<th>Monitor wants to observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>Variable values</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>Objects and Types</td>
</tr>
<tr>
<td>Disk Data</td>
<td>File system and Files</td>
</tr>
<tr>
<td>Hardware Events</td>
<td>Interrupt/Exceptions</td>
</tr>
<tr>
<td>I/O Data</td>
<td>Packets, Buffers</td>
</tr>
</tbody>
</table>
Semantic Gap: A challenge for VMI
Semantic Gap: A challenge for VMI

VMI Challenge: Bridge the semantic gap
Semantic Gap: A challenge for VMI

VMI Challenge: Bridge the semantic gap even for compromised guest

Compromised Guest OS
VMI: Rootkit Detection Technique

- VMI is building block for layered security
  - Trusted hypervisor monitors less trusted guest

Common VMI goal:
- List processes in guest and identify malicious ones

Rootkit goal:
- Confuse VMI & hide malicious process
Rootkit Attack Techniques

- Write text Segment
- Kernel Object Hooking (KOH)
- Direct Kernel Object Manipulation (DKOM)
- Dynamic Kernel Structure Manipulation (DKSM)
Rootkit Attack Techniques

- Write text Segment
  - Change `call` instruction argument

- Kernel Object Hooking (KOH)

- Direct Kernel Object Manipulation (DKOM)

- Dynamic Kernel Structure Manipulation (DKSM)
Listing all the running processes

C-library call

Userspace

Kernel

sys_getdents

("/proc")

System call

ps -a

vfs_readdir

("/proc")

VFS call

proc_root_file->f_op->readdir

FS specific call

proc_root_file ->f_op

*readdir()

Return pids of all processes from list
Listing all the running processes

Userspace

Kernel

C-library call

readdir ("/proc")

sys_getdents ("/proc")

vfs_readdir ("/proc")

proc_root_file->f_op->readdir

proc_root_file->f_op

*readdir()

System call

ps -a

VFS call

FS specific call

proc_root_readdir()

Return pids of all processes from list
Change Text Section (1/2)

```c
vfs_readdir()
{
    ...
}
```

```c
sys_getdents()
{
    ...
    CALL vfs_readdir;
    ...
}
```
vfs_readdir()
{
    ...
}

sys_getdents()
{
    ...
    CALL mal_readdir;
    ...
}
vfs_readdir()
{
    ...
}

sys_getdents()
{
    ...
    CALL mal_readdir;
    ...
}
Listing all the running processes

C-library call

ps -a

System call

sys_getdents ("/proc")

VFS call

vfs_readdir ("/proc")

FS specific call

proc_root_file->f_op->readdir

proc_root_file ->f_op

*readdir()

Return pids of all processes from list

Userspace

Kernel
Listing all the running processes

C-library call

sys_getdents ("/proc")

System call

ps -a

vfs readdir ("/proc")

VFS call

proc_root_file->f_op->readdir

FS specific call

proc_root_file->f_op

*readdir()

Return pids of all processes from list
syscall_nr_getdents = 141
syscall_nr_open = 5

getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];

sys_open()
{
    ...
}

sys_getdents()
{
    ...
}
syscall_nr_getdents = 141
syscall_nr_open = 5
getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];

sys_open()
{
    ...
}

sys_getdents()
{
    ...
}
syscall_nr_getdents = 141
syscall_nr_open = 5

getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];

sys_open()
{
    ...
}

sys_getdents()
{
    ...
}
**Attack**: Change control flow by writing text segment
**Write Text Segment Trust**

**Attack**: Change control flow by writing text segment

- System call table or interrupt descriptor table

- 

- 

- 

- 

- 

- 

- 

-
Write Text Segment Trust

**Attack:** Change control flow by writing text segment
- System call table or interrupt descriptor table

**Defense:** Hypervisor enforced $W \oplus X$

..
Write Text Segment Trust

**Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table

**Defense**: Hypervisor enforced $W \oplus X$
  - $W \oplus X$: All pages writable or executable not both
  - Prevent guest from overwriting executable code pages
  - State of Art: SecVisor[8]
Write Text Segment Trust

**Attack**: Change control flow by writing text segment
- System call table or interrupt descriptor table

**Defense**: Hypervisor enforced $W \oplus X$
- $W \oplus X$: All pages writable or executable not both
- Prevent guest from overwriting executable code pages
- State of Art: SecVisor[8]

**Trust Assumption**: Initial text segment benign
Write Text Segment Trust

**Attack:** Change control flow by writing text segment
- System call table or interrupt descriptor table

**Defense:** Hypervisor enforced $W \oplus X$
- $W \oplus X$: All pages writable or executable not both
- Prevent guest from overwriting executable code pages
- State of Art: SecVisor[8]

**Trust Assumption:** Initial text segment benign

*Cannot prevent attacks on control data in data segment*
Rootkit Attack Techniques

- Write text Segment
  - Change call instruction argument

- Kernel Object Hooking (KOH)

- Direct Kernel Object Manipulation (DKOM)

- Dynamic Kernel Structure Manipulation (DKSM)
Rootkit Attack Techniques

- Write text Segment
  - Change call instruction argument

- Kernel Object Hooking (KOH)
  - Change function pointers (data segment)

- Direct Kernel Object Manipulation (DKOM)

- Dynamic Kernel Structure Manipulation (DKSM)
Listing all the running processes

- **C-library call**
  - readdir ("/proc")
  - System call
    - sys_getdents ("/proc")
  - VFS call
    - vfs_readdir ("/proc")
  - FS specific call
    - proc_root_file->f_op->readdir
    - proc_root_file->f_op
    - *readdir()
    - proc_root_readdir()
    - Return pids of all processes from list

- Userspace
- Kernel
Listing all the running processes

C-library call

Userspace

Kernel

ps -a

System call

sys_getdents ("/proc")

VFS call

vfs_readdir ("/proc")

FS specific call

proc_root_file->f_op->readdir

Return pids of all processes from list

proc_root_file->f_op

*readdir()
Kernel Object Hooking (KOH)

```c
struct file{
    ...
    struct path f_path;
    const struct file_operations *
f_op;
    struct fown_struct f_owner;
    void *f_security;
    fmode_t f_mode;
    off_t f_pos;
    ...
}

file struct object for /proc
```

```c
struct file_operations{
    ...
    int (*open) (struct inode *, struct file *);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    ...
}
```

```c
file_operations object for /proc
```

```c
proc_root_readdir(){
    ...
}
```

```c
readdir code for proc filesystem
```
Kernel Object Hooking (KOH)

```c
struct file{
    ...
    struct path f_path;
    const struct file_operations *
        *f_op;
    struct fown_struct f_owner;
    void *f_security;
    fmode_t f_mode;
    off_t f_pos;
    ...
}
```

file struct object for /proc

```c
struct file_operations{
    ...
    int (*open) (struct inode *, struct file *);
    ssize_t (*read) (struct file *, char __user *,
                     size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    ...
}
```

file_operations object for /proc

```c
proc_root_readdir() {
    ...
}
```

readdir code for proc filesystem
Kernel Object Hooking (KOH)

```c
struct file{
    ...
    struct path f_path;
    const struct file_operations *
        *f_op;
    struct fown_struct f_owner;
    void *f_security;
    fmode_t f_mode;
    off_t f_pos;
    ...
}
```

- file struct object for /proc

```c
struct file_operations{
    ...
    int (*open) (struct inode *, struct file *);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    ...
}
```

- file_operations object for /proc

```c
proc_root_readdir(){
    ...
}
```

- readdir code for proc filesystem

```c
mal_readdir(){
    ...
}
```

- malicious readdir code for proc filesystem
KOH Defense Trust Assumptions

- **Attack**: Change function pointers in objects
- **Defense**: Protect initialized function pointers
  - Redirect hooks to write protected memory
  - State of Art: HookSafe[9]
- **Trust Assumption**: Pristine initial OS copy
  - Administrator can white-list safe modules
  - All hooks are learned during dynamic analysis

Cannot protect data fields in heap section
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument

- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)

- **Direct Kernel Object Manipulation (DKOM)**

- **Dynamic Kernel Structure Manipulation (DKSM)**
Rootkit Attack Techniques

- **Write text Segment**
  - Change *call* instruction argument

- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)

- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants

- **Dynamic Kernel Structure Manipulation (DKSM)**
Direct Kernel Object Manipulation

Guest Kernel Heap

01010011
11110010
11010110
01101010

10111001
11010100
10011100
10101011

Linux Guest OS

Hypervisor
Direct Kernel Object Manipulation

Process list used to enumerate

Process tree used by scheduler

Linux Guest OS

Hypervisor

Guest Kernel Heap

01010011 11110010 11010110 01101010 10111001 11010100 10011100 10101011
Direct Kernel Object Manipulation

Process list used to enumerate

Process tree used by scheduler

Guest Kernel Heap

Linux Guest OS

Hypervisor

Stony Brook University
Direct Kernel Object Manipulation

- **Process list used to enumerate**
- **Process tree used by scheduler**
Direct Kernel Object Manipulation

Process list used to enumerate

Process tree used by scheduler

Guest Kernel Heap

01010011
11110010
11010110
01101010
10111001
11010100
10011100
10101011
Direct Kernel Object Manipulation

Process list used to enumerate

Process tree used by scheduler

Guest Kernel Heap

01010011
11110010
11010110
01101010
10111001
11010100
10011100
10101011

Linux Guest OS

Hypervisor

Stony Brook University
Direct Kernel Object Manipulation

Invariant: Set of processes in Scheduler tree = Process list

Process list used to enumerate

Process tree used by scheduler
Direct Kernel Object Manipulation

Process list used to enumerate

Invariant: Set of processes in Scheduler tree = Process list

Process tree used by scheduler

Guest Kernel Heap

Linux Guest OS

Hypervisor

Invariant: Set of processes in Scheduler tree = Process list
Direct Kernel Object Manipulation

Invariant: Set of processes in Scheduler tree = Process list

Process list used to enumerate

Process tree used by scheduler

Guest Kernel Heap

Hypervisor

Linux Guest OS

Scheduler tree

Invariant: Set of processes in Scheduler tree = Process list

Stony Brook University
Direct Kernel Object Manipulation

Invariant: Set of processes in Scheduler tree = Process list

Invariant Violated!!!

Process list used to enumerate

Process tree used by scheduler
**Direct Kernel Object Manipulation**

Assume attacker can’t win race with asynchronous checking

**Invariant: Set of processes in Scheduler tree = Process list**

**Invariant Violated!!!**
**DKOM Defense Trust Assumptions**

- **Attack**: Modify the kernel objects in heap
- **Defense**: Asynchronously check invariants
  - State of Art: OSck[10]
- **Trust Assumption**:
  - All security invariants can be learned
  - Invariants can be checked in single search
  - Attackers cannot win races with the monitor
  - Availability of other integrity defenses

Cannot prevent attacks or detect transient attacks
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument

- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)

- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants

- **Dynamic Kernel Structure Manipulation (DKSM)**
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument

- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)

- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants

- **Dynamic Kernel Structure Manipulation (DKSM)**
  - Change data structure interpretation
Data Structure Manipulation Attack

Hypervisor

Linux Guest OS

Guest Kernel Heap

Stony Brook University
Data Structure Manipulation Attack

```
struct task_struct{
    ......
    char comm[16];
    void *notifier;
    ......
}
```

`strlcpy(tsk->comm, buf, 16);`

Initialization

Process Command
Data Structure Manipulation Attack

```
struct task_struct{
    char comm[16];
    void *notifier;
    ...
};
```

Hypervisor
- Firefox
- Skype

Guest Kernel Heap
- Typecast & offset math

Linux Guest OS
- strlcpy(tsk->comm, buf, 16);

Initialization
- Offset 1128
- Process Command

Stony Brook University
Data Structure Manipulation Attack

```c
struct task_struct{
  char comm[16];
  void *notifier;
  ...
};
```

```
strcpy(tsk->comm, buf, 16);
```

Hypervisor

- Firefox
- Skype

Typecast & offset math
Data Structure Manipulation Attack

```
struct task_struct{
    char comm[16];
    void *notifier;
    ...
};
```

Guest Kernel Heap
```
strlcpy(tsk->comm, buf, 16);
```

Hypervisor
- Firefox
- Skype
- Adore-ng

Typecast & offset math

Offset Process Command
1128 Initialization

Data Structure Manipulation Attack

```
struct task_struct{
    char comm[16];
    void *notifier;
    ......
}
```

**Initialization**

```c
strlcpy(tsk->comm,"Chrome",16);
notifier = kmalloc(16,GFP_KERNEL);
strlcpy(tsk->notifier, buf, 16);
```

**Offset**

```
1128
```

**Process Command**

- Hypervisor
  - Firefox
  - Skype
  - Adore-ng

**Malicious Guest OS**

**Guest Kernel Heap**

**Change field interpretation**

**Typecast & offset math**

- Guest Kernel Heap
- Linux Guest OS
Data Structure Manipulation Attack

struct task_struct{
    char comm[16];
    void *notifier;
    ...
}

Initialization

Hypervisor

Malicious Guest OS

Guest Kernel Heap

strlcpy(tsk->comm,"Chrome",16);
notifier = kmalloc(16,GFP_KERNEL);
strlcpy(tsk->notifier, buf, 16);

Typecast & offset math

All processes are benign

Stony Brook University
Data Structure Manipulation Attack

```
struct task_struct{
    char comm[16];
    void *notifier;
    ...
}
```

**Malicious Guest OS**

**Offset**

1128

**Process Command**

```
strcpy(tsk->comm,"Chrome",16);
notifier = kmalloc(16,GFP_KERNEL);
strcpy(tsk->notifier, buf, 16);
```

**Initialization**

**Hypervisor**

- = Chrome
- = Chrome
- = Chrome

**Typecast & offset math**

**Change field interpretation**

**All processes are benign**

Malicious or Compromised OS can violate VMI assumptions
DKSM Trust Assumptions

➢ **Assumption:** Consistent structure interpretation

➢ **Attack:** Change interpretation of a data structure
  ▪ Mislead VMI tools by presenting false system state

➢ **Defense:** No existing defense
  ▪ CFI on benign kernel may help prevent bootstrapping
  ▪ Attack obviated by generous threat models
  ▪ Trust guest OS to be uncompromised and benign

Structure manipulation: Realistic but outside threat model
Recap

- **Weak Semantic Gap: Solved**
  - An engineering challenge
  - Assume guest OS respects data structure templates

- **Strong Semantic Gap: Open problem**
  - Malicious or Compromised OS
  - Exploit fragile assumptions to confuse VMI designs

Problem worth working: Strong Semantic Gap
Many VMI designs are fairly expensive
  - Some run sibling VM on dedicated core for analysis

VMI can be useful in a cloud or multi-VM system
  - Manage overhead & scalability with increase in VMs

Some VMI systems trade risk to reduce overhead
  - Identify techniques to minimize both overheads & risk

One consideration for VMI systems: Scalability
Privacy

- VMI can create new side-channels in cloud
  - Scan period or sibling VM activities using cache timing
- Shouldn’t force choice of integrity or privacy risks.
- VMI should evaluate risks of new side channels.
  - Take into account compliance regulation

Another consideration for VMI systems: Privacy
References


