Detecting Exploit Code Execution in Loadable Kernel Modules

Haizhi Xu, Wenliang Du, Steve J. Chapin
Systems Assurance Institute Syracuse University, NY USA
IEEE ACSAC 2004
Presented by Yu-Han Li
Outline

- Introduction
- Overview of HECK
  - Restrictions on the kernel modules
  - The architecture of HECK
- Implementation of HECK
- Overhead measurement
- Conclusion
Introduction

Loadable kernel modules (LKM) have unrestricted access to all kernel memory and I/O space
- Kernel-module exploitation can jeopardize the system
- Hardware protection doesn’t isolate kernel modules from the core kernel

Our approach
- Hybrid Extension Checker for Kernels (HECK)
  - detect kernel modules for malicious code execution
Overview of HECK

- Work is based on three assumptions:
  - There are *unexpected* implementation error in kernel modules
  - The attacker has *no root privilege* but authorized to operate on a device
  - The source code of modules are *trusted*,
    - compiled with a trusted compiler
    - binary images are stored safely
Overview of HECK

- Restrictions on the kernel modules
  - Sensitive instructions are critical to exploit code execution
    - branch, memory access instruction
  - Isolate the module except for operations based on specification of legitimate request
  - Prevent the module from execution code on the *kernel stack* or other *data section*
Overview of HECK

- Perform restrictions on modules by run-time
  - **R1**: monitor “branch instructions”
    - Inside the module
Overview of HECK

- Outside the module
  - We extract the legitimate external calls to the kernel functions from static analysis

- R2: don’t give module unnecessary permissions access to the kernel data
  - Disallow write access to the stack outside the local frame
Overview of HECK

- Trust kernel structure introduced by permitted kernel function
  - If kernel function calls a module function, allow module function to access the parameter
  - Allow access to dynamically allocated resources created by the module
Overview of HECK

The architecture of HECK

A specification generation part

Permission lists

A run-time monitoring part

Run-time checking code

Dynamic resources

1. Function call parameter
2. Dynamically allocated memory
3. Dynamically memory mapped I/O
4. Return address

The Hecked module

Script to convert into ASM

Gen. spec. using OBJDUMP

Hooks for registering
1. At entrance
2. After returning from allocation fun.

1. Run-time checking at check point
2. Resource registered by activation of registration function
Overview of HECK

- Original module vs HECKed module

The original module
- R.O. data section
- R.W. data section
- code section

The HECKed module
- R.O. data section
- static permission lists
- R.W. data section
- dynamic permission lists
- code section
- Dynamic checking functions
- Dynamic specification management functions
Implementation of HECK

Description of the specification

- **Specification** – describe permissions granted to a module
- **Permission** – an operation on a block of virtual memory, I/O ports, marked by starting and ending address
- Five pairs of permission lists
  - allow_mem_read (neverallow)
  - allow_mem_write (neverallow)
  - allow_mem_call (neverallow)
Implementation of HECK

- allow_IO_read (neverallow)
- allow_IO_write (neverallow)

At each checking point, choose one pair of allow/neverallow list

The following shows a fraction of the static specificati
generated for ide-cc, the Linux IDE CDROM driver.

```
# default: neverallow stack call
# default: neverallow data_sections call
allow kmalloc_Rsmp_93d4cfe6 call
allow kfree_Rsmp_037a0cbe call
allow printk_Rsmp_1b7d4074 call
allow sprintf_Rsmp_1d26aa98 call
allow atapi_output_bytes_Rsmp_aaffe3a66 call
... ...

# IDE 1
allow IO_port 0x170--0x177 read-write
allow IO_port 0x376--0x376 read-write
# Bus master IDE
allow IO_port 0xda000--0xda0f read-write
neverallow proc_root_Rsmp_020bc977 write
```
Implementation of HECK

"Run-Time checking"

Step 1: statically, inject a checking hook before every sensitive instruction

Step 2: before executing the sensitive instruction, the hook will call the corresponding monitor subroutine

Step 3: if the checking code finds a violation, it will call the corresponding handler function

The original code

```assembly
.type cdrom_log_sense, @function

cdrom_log_sense:
    subl $8, %esp
    movl 20(%esp), %edx
    movl %ebx, 4(%esp)
    xorl %ebx, %ebx
    movl 16(%esp), %eax
    testl %edx, %edx
    je .L122
    testl %eax, %eax
    je .L122
    movl 12(%eax), %eax
    testl %eax, %eax
    je .L121
.L122:
    xorl %eax, %eax
.L120:
    movl 4(%esp), %ebx
    addl $8, %esp
    ret
```

## dcheck ret!
```
push %ebx
movl 4(%esp), %ebx
call d_policy_fret
pop %ebx
```
Implementation of HECK

1. Search never allow list
2. Search allow static and dynamic list
3. Otherwise violate the specification
Implementation of HECK

- Dynamic resource management

Code under monitoring

```assembly
... ...
.type cdrom_log_sense, @function
	#xhz d_policy function start!
	cdrom_log_sense:
		call d_policy_fenter
		subl $8, %esp
	### dcheck Optimization--no need for dcheck
	#original instruction 7:
		movl 20(%esp), %edx
	### dcheck Optimization--no need for dcheck
	#original instruction 8:
		movl %ebx, 4(%esp)
		xorl %ebx, %ebx
```

Functions for the dynamic resource registration

- `d_policy_fenter`
- `d_param_ide_device_info_reg`
- `allow_mem_read`
- `allow_mem_write`
- `allow_mem_call`

Dynamic permission lists:
Implementation of HECK

- Instructions manipulating local variables and pc-

```
... ...
.type cdrom_log_sense,@function
.xhz d_policy function start!
cdrom_log_sense:
    call d_policy_enter
    subl $8, %esp
### dcheck Optimization--no need for dcheck
#original instruction 7:
    movl 20(%esp), %edx
### dcheck Optimization--no need for dcheck
#original instruction 8:
    movl %ebx, 4(%esp)
    xorl %ebx, %ebx
### dcheck Optimization--no need for dcheck
#original instruction 9:
    movl 16(%esp), %eax
    testl %edx, %edx
    je .L122
    testl %eax, %eax
    je .L122
### dcheck indirect read!
    push %ebx
    lea 12(%eax), %ebx
    call dcheck_EBP_r
    pop %ebx
### original instruction 10:
    movl 12(%eax), %eax
    testl %eax, %eax
    jo .L121
```
Overhead measurement

Platform
- Pentium II 400MHZ dual-processor SMP
- Red-Hat Linux 8.0
- 40Xmax ATAPI CDROM drive
- 10/100M PCI network interface card
- Kernel version 2.4.18-14smp
Overhead measurement

Space overhead of run-time checking code

<table>
<thead>
<tr>
<th></th>
<th>number of instructions</th>
<th>number of checking points excluding the local variables</th>
<th>percentage of instructions under check</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdrom.o</td>
<td>4879</td>
<td>717</td>
<td>15%</td>
</tr>
<tr>
<td>ide.cd.o</td>
<td>4749</td>
<td>814</td>
<td>17%</td>
</tr>
<tr>
<td>8139too.o</td>
<td>2850</td>
<td>466</td>
<td>16%</td>
</tr>
<tr>
<td>fat.o</td>
<td>27979</td>
<td>1695</td>
<td>6%</td>
</tr>
<tr>
<td>ext3.o</td>
<td>60534</td>
<td>3401</td>
<td>6%</td>
</tr>
<tr>
<td>total</td>
<td>100991</td>
<td>7093</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table 1. number of checking points in the modules: In the experiment that removes local variables checking, we reduces the number of checking points by moving the local variables checking to the static checking procedure. In this case, 6%-17% (on average 7%) of all the instructions need run time checking.
Overhead measurement

Run-time overhead on CPU intensive

Figure 4. Run-time overhead on a CPU intensive application micro-benchmark, serving as the upper bound for the device drivers
Overhead measurement

- Runtime overhead on some I/O intensive app
Conclusion

- We propose an approach to detect in-module exploit code execution on instruction level.
- HECK works on binary code level.
- Our tool detects a large set of malicious code execution with modules.