Countering Kernel Rootkits with Lightweight Hook Protection

Zhi Wang, Xuxian Jiang, Weidong Cui, Peng Ning

(From the proceeding of the 16th ACM CCS 2009)

Presented by: Nathan Sriboonlue

Outline

Introduction: Rootkit

- Rootkit is a software program designed to gain control over a system or network.
- Rootkits can not only hide their presence but also tamper with OS functionalities to launch various attacks.
  - Opening backdoors
  - Stealing private information
  - Escalating privileges of malicious processes
  - Disable defense mechanisms

Previous works

- Three major research categories:
  - Analysis of rootkit behavior
    - *Panorama, HookFinder, K-Tracer, and PoKeR*
  - Detection of rootkits based on certain symptoms exhibited by rootkit infection
    - *Copilot, SBCFI, and Vmwatcher*
  - Preservation of kernel code integrity by preventing malicious rootkit code from executing
    - *SecVisor, Patagonix, and NICKLE.*

Motivation

- The act of preventing malicious rootkit codes from executing alone is not enough. This type of security can be bypassed easily.
- Rootkits such as the return-oriented ones, will first subvert kernel control flow and then launch the attack by only utilizing legitimate kernel code snippets.
  - Hijacking attack on return address and function pointers

Motivation

- In addition to the preservation of kernel code integrity, it is also equally important to safeguard relevant kernel control data
- By preserving the kernel control flow integrity, it enables the system to block out all rootkit infections in the first place.

Kernel Hook

- As there has been extensive research on the protection of return address, this paper is solely focused on the protection of function pointers.
- Function pointers are typically hijacked or “hooked” by rootkits, thus for ease of presentation, the paper addressed the term function pointers and kernel hooks interchangeably.

Challenge(1)

- In OS such as windows and linux, there exist thousands of kernel hooks and these kernel hooks can be widely scattered across the kernel space.
- To monitor all write to these system pages would introduce significant performance overhead.
- Extremely inefficient as their previous study showed that only 1% of kernel memory writes may cause problem.

Ubuntu Study

- Analysis of a typical Ubuntu 8.04 server by using a whole-system emulator called QEMU.
- Within a randomly-selected 100 secs, there were 700,970,160 total kernel memory writes.
- Only 6,479,417(1%) memory writes were possible to create page fault.

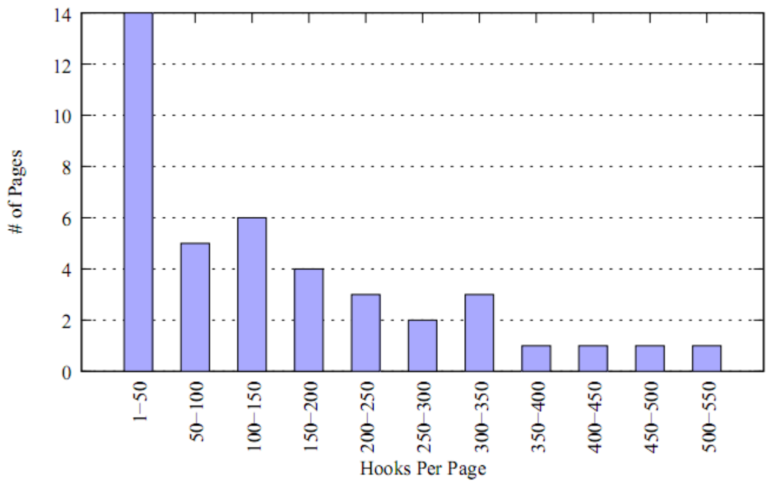


Figure 1: Distribution of $5,881$ **kernel hooks in a running Ubuntu system**

Challenge(2)

- Protection Granularity Gap
  - effective protection requires byte-level granularity while commodity computers allow only for protection at a much broader page level.

HookSafe

- HookSafe is a hypervisor-based lightweight system that aims to achieve large-scale protection of kernel hooks in a guest OS.
- HookSafe solves the protection granularity gap problem by creating a shadow copy of the kernel hooks in a centralized location.
- Any attempt to modify the shadow copy will be trapped and verified by the underlying hypervisor while the regular read access will be simply redirected to the shadow copy.

HookSafe

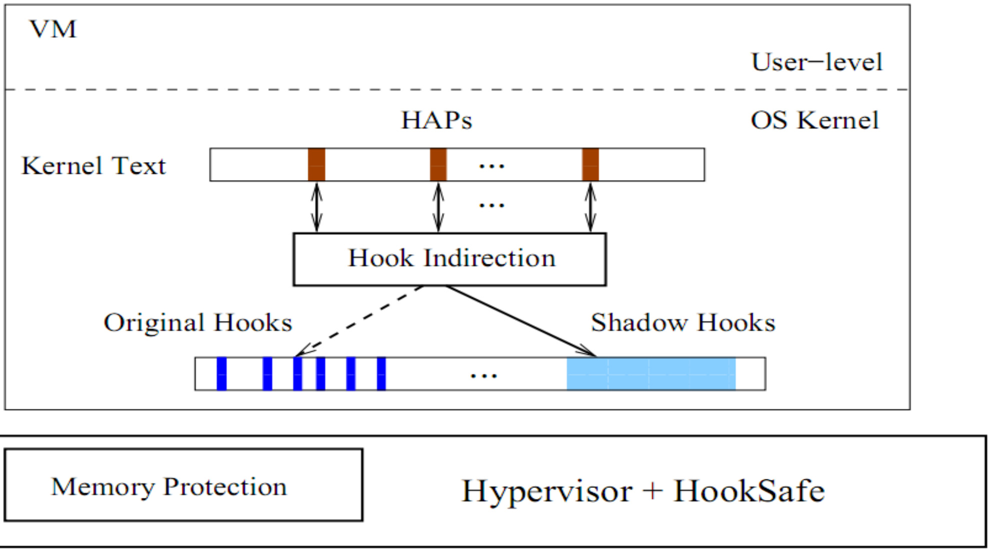


Figure 3: The architecture of online hook protection.

Implementation

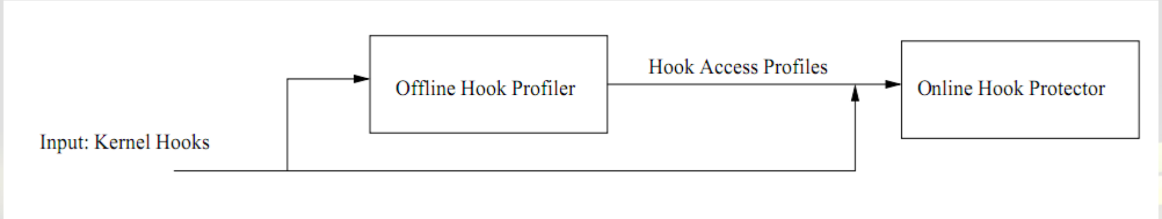


Figure 2: The HookSafe architecture

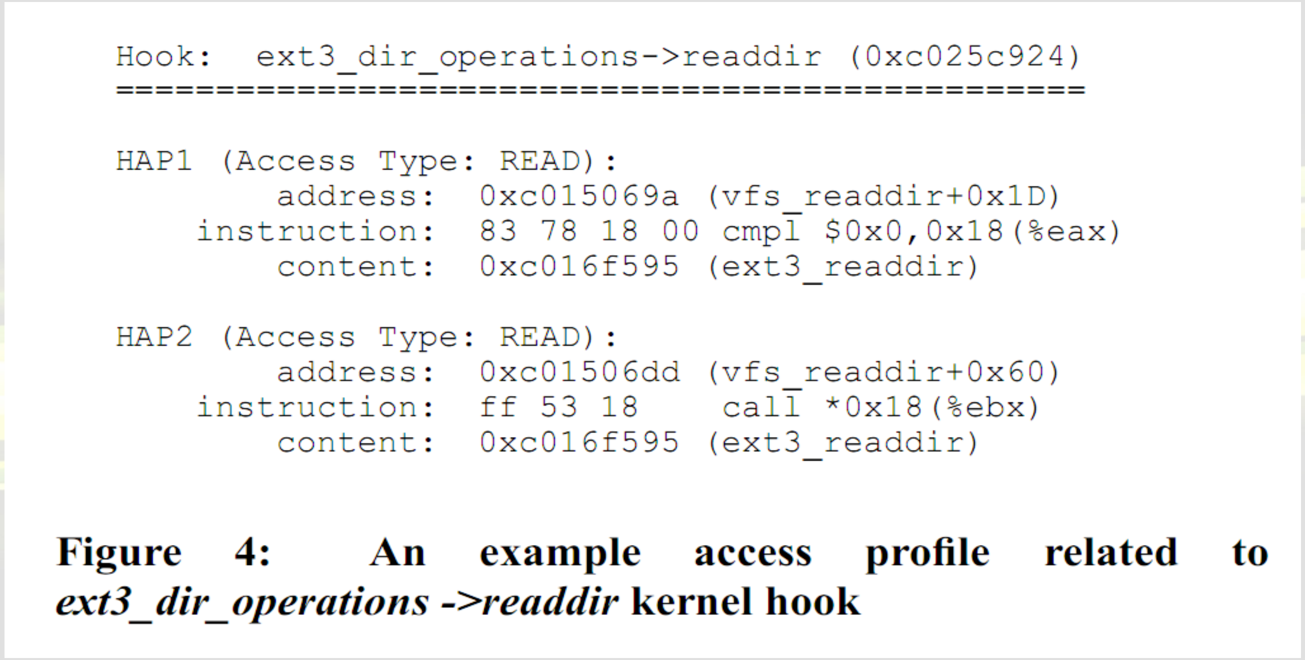
- Given a set of kernel hooks for protection, HookSafe achieves its functionality in two key steps:
  - Offline Hook Profiler
  - Online Hook Protector

Offline Hook Profiler

- Offline hook profiler takes in kernel hook inputs and profiles them into a hook access files.
- These hook access files contain information such as access type and the values associated with it.
- These instructions that access a hook are known as Hook Access Points (HAPs).

Offline Hook Profiler

•



```
Hook:  ext3_dir_operations->readdir (0xc025c924)
===================================================

HAP1 (Access Type: READ):
        address:  0xc015069a (vfs_readdir+0x1D)
    instruction:  83 78 18 00 cmpl $0x0,0x18(%eax)
        content:  0xc016f595 (ext3_readdir)

HAP2 (Access Type: READ):
        address:  0xc01506dd (vfs_readdir+0x60)
    instruction:  ff 53 18    call *0x18(%ebx)
        content:  0xc016f595 (ext3_readdir)
```

Figure 4: An example access profile related to *ext3_dir_operations ->readdir* kernel hook

Online Hook Protector

- Taking hook access profiles as input, online hook protector creates a shadow copy of all protected hooks and instruments HAP instructions such that their accesses will be transparently redirected to the shadow copy.
- Shadow hooks are aggregated together in a central location and protected from any unauthorized modification.

Online Hook Protector

- To reduce performance overheads, HookSafe handles read and write differently.
- Write - transfer control from guest kernel to the hypervisor, update the memory, then return control back to the guest kernel.
- Read - use a piece of indirection code residing in the guest OS kernel memory to read the corresponding shadow hook.

- **Note:** read accesses are much more common than write, thus we benefit greatly by keeping the command under guest kernel level.

Online Hook Protector

- When read is performed, HookSafe will do a consistency check between the original kernel hook and its shadow copy to make sure that it has not been compromised.
- To validate write request, HookSafe requires the new hook value to be seen in the offline profiling phase. Other common techniques can also be applied here:
  - Valid code region
  - Valid function value type
- Once the write request has been validated as legitimate, both shadow copy and its original hook are updated.

Evaluation

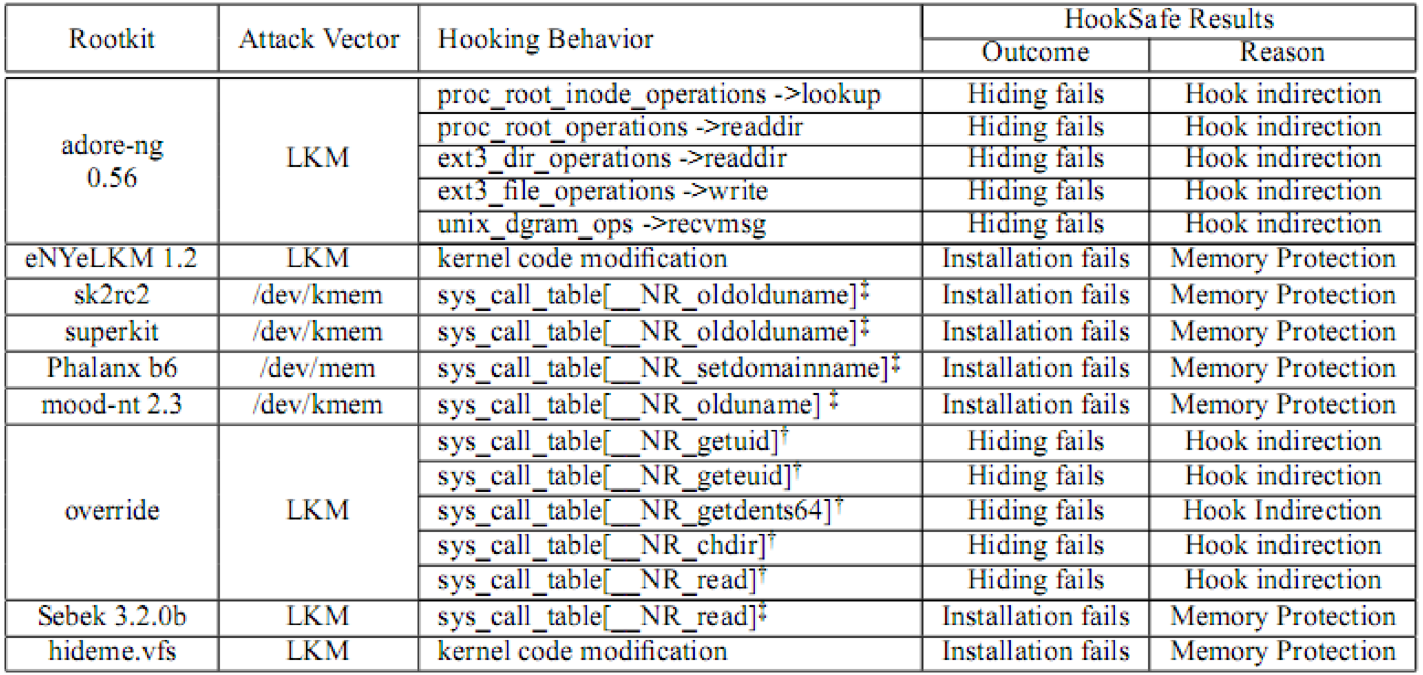
- Two sets of experiments:
  - Evaluation of HookSafe's effectiveness against nine real-world rootkits
  - Evaluation of performance overhead introduced by HookSafe on benchmark programs and real-world application

Set up

- HookSafe takes in two sets of kernel hooks input:
  - The first set contains 5,881 kernel hooks in preallocated memory areas of main linux kernel and dynamically loaded kernel modules
  - The second set is from 39 kernel objects that will be dynamically allocated from kernel heap.
- These hook inputs are obtained through scanning of the data/bss sections of the kernel and LKMs in a guest VM running Ubuntu Server 8.04.

Effectiveness

- Nine state-of-the-art kernel rootkit:
  - Adore-ng, eNYeLKM 1.2, sk2rc2, superkit, Phalanx b6, mood-nt 2.3, override, Sebek 3.2.0b, and hideme.vfs
- HookSafe successfully prevented all of the rootkits tested from modifying the protected kernel hooks.



Rootkit	Attack Vector	Hooking Behavior	HookSafe Results	
			Outcome	Reason
adore-ng 0.56	LKM	proc_root_inode_operations ->lookup	Hiding fails	Hook indirection
		proc_root_operations ->readdir	Hiding fails	Hook indirection
		ext3_dir_operations ->readdir	Hiding fails	Hook indirection
		ext3_file_operations ->write	Hiding fails	Hook indirection
		unix_dgram_ops ->recvmsg	Hiding fails	Hook indirection
eNYeLKM 1.2	LKM	kernel code modification	Installation fails	Memory Protection
sk2rc2	/dev/kmem	sys_call_table[__NR_oldolduname]‡	Installation fails	Memory Protection
superkit	/dev/kmem	sys_call_table[__NR_oldolduname]‡	Installation fails	Memory Protection
Phalanx b6	/dev/mem	sys_call_table[__NR_setdomainname]‡	Installation fails	Memory Protection
mood-nt 2.3	/dev/kmem	sys_call_table[__NR_olduname] ‡	Installation fails	Memory Protection
override	LKM	sys_call_table[__NR_getuid]†	Hiding fails	Hook indirection
		sys_call_table[__NR_geteuid]†	Hiding fails	Hook indirection
		sys_call_table[__NR_getdents64]†	Hiding fails	Hook Indirection
		sys_call_table[__NR_chdir]†	Hiding fails	Hook indirection
		sys_call_table[__NR_read]†	Hiding fails	Hook indirection
Sebek 3.2.0b	LKM	sys_call_table[__NR_read]‡	Installation fails	Memory Protection
hideme.vfs	LKM	kernel code modification	Installation fails	Memory Protection

Table 1: Effectiveness of HookSafe in preventing 9 real world kernel rootkits: *Hiding fails* indicates that although the rootkit modified the original kernel hooks, it failed to hijack the control flow because HookSafe has redirected the hook to its shadow; *Installation fails* indicates that the rootkit hooking behavior causes the memory protection violation, hence failing the installation. Additionally, depending on how the rootkit searches for the system call table, it may locate either the original system call table (marked with †) or the shadow system call table (marked with ‡).

Performance

- To evaluate performance overhead introduced by HookSafe, they measured the runtime overhead over 10 computer tasks.
- Set up:
  - Dell Optiplex 740
  - AMD64 X2 5200+
  - 2GB memory
  - Xen Hypervisor 3.3.0
  - Ubuntu server 8.04

Performance

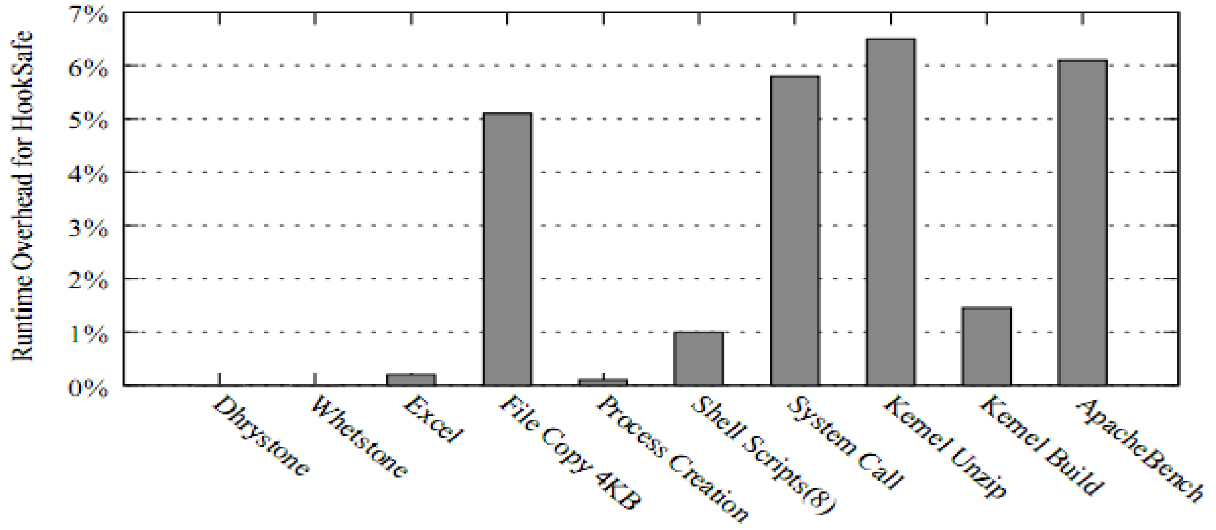


Figure 8: Runtime overhead on UnixBench and three application benchmarks for HookSafe.

Contribution

- HookSafe is the first system that is proposed to enable large-scale hook protection with low performance overhead.
- Extremely credible as it performed well against various advance rootkits.
- Overcame the critical challenge of the protection granularity gap.

Weakness

- Construction of hook access profiles may be incomplete.
- Small time lag before detection of inconsistencies between the original kernel hooks and their shadow copies.
- Need prior knowledge of the set of kernel hooks that need to be protected

Future Work

- Perform both dynamic analysis and static analysis of the source code to improve the coverage of finding all the HAPs.
- Combine HookSafe with hook finding applications such as HookFinder and HookMap

Reference

Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proc. Of the 16th ACMConference on Computer and Communication Security (CCS 2009)*, Chicago, IL, October 2009.

Question?