Kernel Rootkit Protection

Andy Podgurski
EECS Dept.
Case Western Reserve University

Sources

- *A Case Study of the Rustock Rootkit and Spam Bot*, K. Chiang and L. Lloyd, 2007.
- *Tracking Rootkit Fingerprints with a Practical Memory Analysis System*, W. Cui et al, USENIX, 2011.
- *Rootkits*, G. Hoglund and J. Butler, Addison-Wesley, 2006.
- *Multi-Aspect Profiling of Kernel Rootkit Behavior*, R. Riley et al, EuroSys 2009.
- *Countering Persistent Kernel Rootkits through Systematic Hook Discovery*, Z. Wang et al, 2008.
- *Countering Kernel Rootkits with Lightweight Hook Protection*, Z. Wang et al, CCS 2009.
- *Windows Rootkits: A Game of Hide and Seek*, S.Sparks et al, Handbook of Security and Networks, 2011

Rootkit

- Set of programs providing a persistent, hard-to-detect presence on a target computer
- Permits remote command and control, eavesdropping, disabling defenses
- It hides code and data from security programs and system utilities
  - e.g., ps, ls, netstat
- Usually requires access to OS kernel
- Typically injected in attack
  - e.g., device driver buffer overflow
- May be used for legitimate surveillance
  - e.g., of computer use by employees

Example Rootkit Components

- File Hider
- Network Operations
- Registry Hider (Windows)
- Process Hider
- Boot Service
- Utilities

Example: Rustock Rootkit and Spam Bot

- Network traffic dump indicated that all command & control (C&C) communications are *encrypted* using RC4
- Two phases: key exchange and instructions to infected host
  - HTTP POSTs

Rustock Command & Control

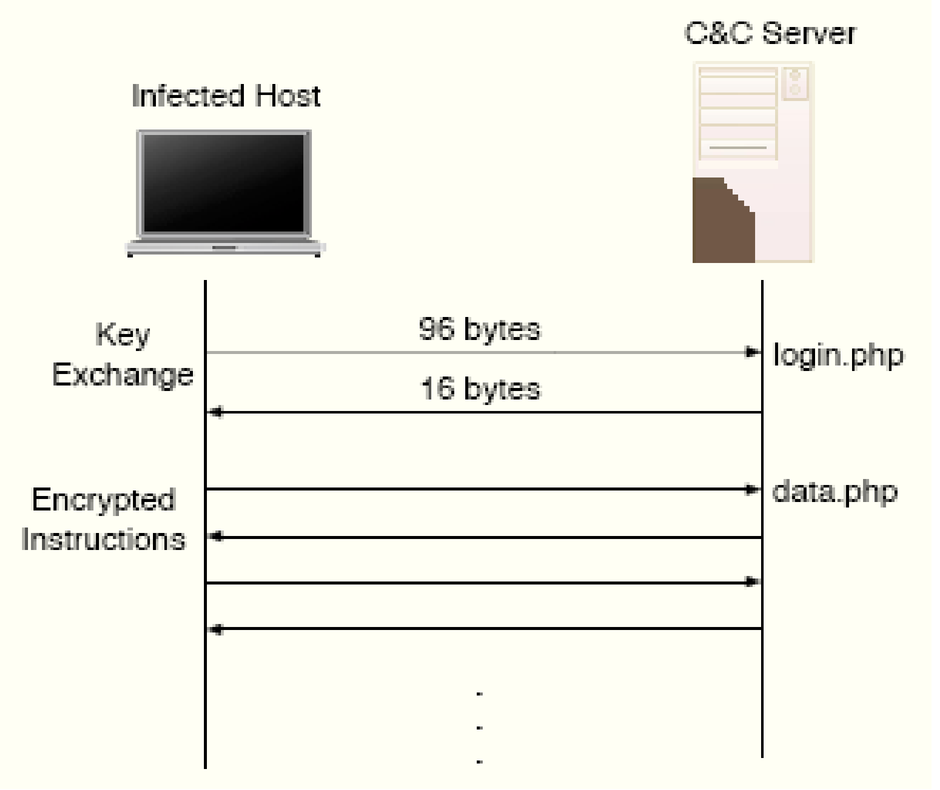


Figure 1: Spam bot command and control

Static Analysis of Rustock

- Applied to obfuscated disassembly code
  - Used IDA Pro 5.0 disassembly tool
- Four main malware components:
  - Initial deobfuscation routine
  - Rootkit loader
  - Rootkit
  - Spam module

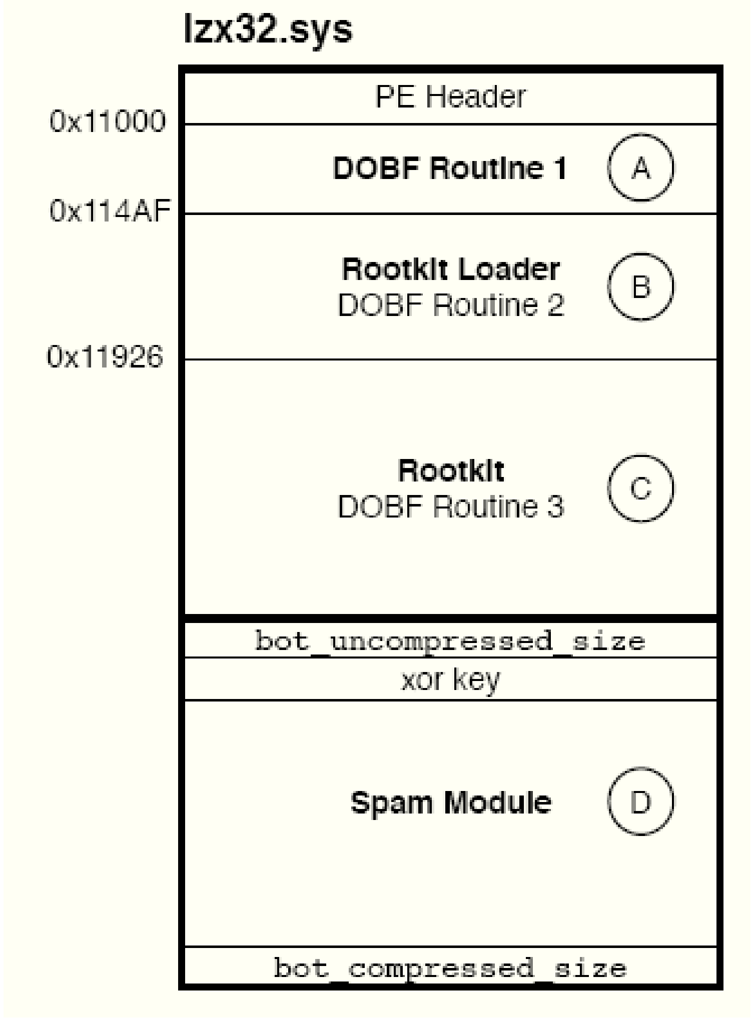


Figure 2: Overview of the lzx32.sys malware. In our analysis we break it down into four parts: A. The first deobfuscation routine, B. The rootkit loader which contains the second deobfuscation routine, C. The rootkit containing the third deobfuscation routine, and D. The spam module.

[Chiang & Lloyd]

Message	Message Contents or Summary
Client 1	"kill.txt"
Server 1	Server response specifies processes to terminate and files to delete from the client
Client 2	Information about the client
Server 2	Information for the client about the client and file names to create or request for subsequent communications with the server
Client 3	"neutral.txt"
Server 3	List of domain names to query for mail servers to use
Client 4	"unlucky.txt"
Server 4	List of SMTP server responses that indicate failure
Client 5	"tmpcode.bin"
Server 5	Binary data that specifies the formatting of spam message to be sent by the client
Client 6	"tmpcode.bin"
Server 6	Binary data including spam content
Client 7	"-"
Server 7	List of target email addresses

Table 1: Summary of decrypted C&C commmnications between the infected client and the server.

[Chiang & Lloyd]

OS Components Compromised by Rootkits (Windows)

- I/O Manager
- Device & file system drivers
- Object Manager
- Security Reference Monitor – responsible for access checking and user privileges
- Process & Thread Manager
- Registry Manager
- Memory Manager

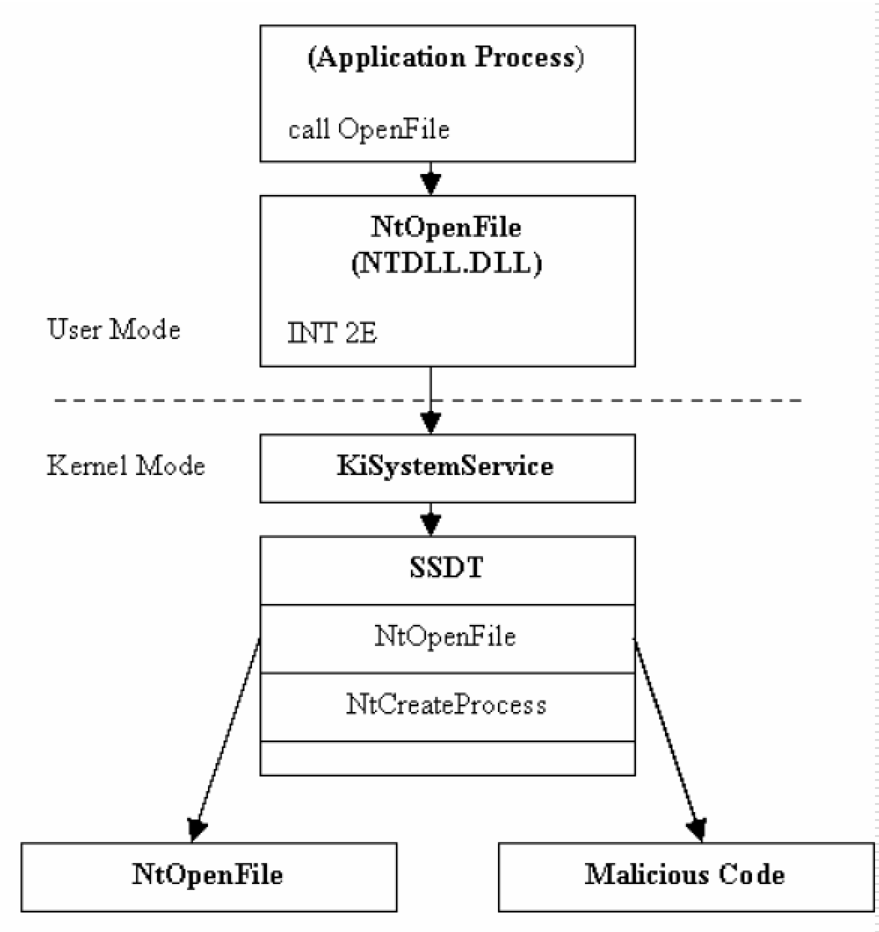
Hooking (Execution Path Redirection)

- A rootkit must either:
  1. Alter execution paths of the OS
  2. Modify kernel data objects
- It can do (1) by “hooking” (overwriting) function code or function-pointers in either
  - A user-mode API
  - The OS kernel

Types of Hooking in Windows

- Patch function pointers:
  - Import/export table hooking
    - Intercepts Win32 library (DLL) calls
  - System service dispatch table hooking
    - SSDT resides in kernel memory
  - Interrupt descriptor table hooking
- Modify binary code of target function:
  - Inline function hooking

System Service Dispatch Table



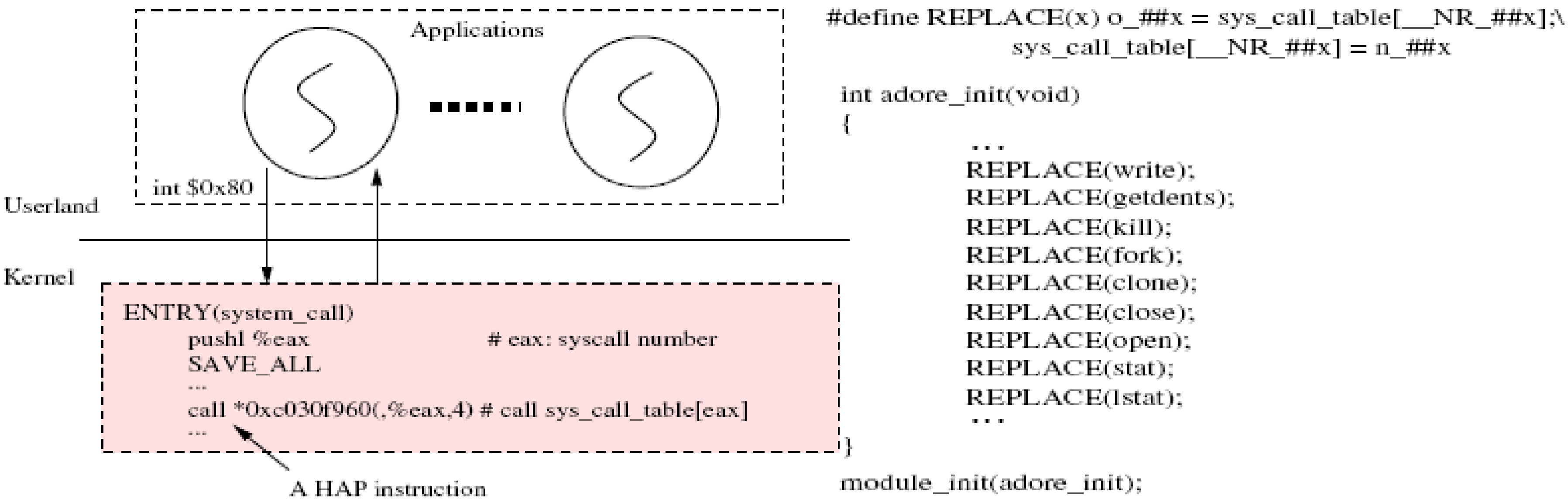
Kernel Hooks

- More powerful and harder to detect than user-mode hooks
  - Places rootkit and protection/detection software at same privilege level
- Rootkit may access kernel memory by implementing a device driver
- Return-oriented rootkits use ROP to subvert kernel control flow
  - Hijacking function pointers or return addresses on stack
  - Using legitimate kernel code snippets
- There may be thousands of hooks widely scattered throughout kernel space

Safeguarding Kernel Hooks

- Kernel code can be marked read-only
- Hence, rootkits now usually implant hooks in kernel data (call and jump target addresses)
  - e.g., system call table
  - called *hook attach* (or *access*) *points* (HAPs)
- One approach is to use HW page protection to monitor writes to kernel hooks
- However, thousands of kernel hooks may be collocated with writeable kernel data
- Trapping all writes to pages with hooks introduces high overhead
- Kernel hook protection requires byte-level granularity

Hook Attach Points (HAPs)



(a) The system call dispatcher on Linux (b) The Linux adore rootkit

Fig. 1. A HAP instruction example inside the Linux system call dispatcher – the associated kernel data hooks have been attacked by various rootkits, including the Linux adore rootkit [1]

[Wang et al, 2008]

HookSafe [Wang et al]

- Focuses on protecting function pointers
- Observation: once initialized kernel hooks rarely change value
- HookSafe relocates kernel hooks to dedicated page-aligned centralized memory location
- HookSafe uses a hook indirection layer to regulate accesses with HW page protection
  - Avoids unnecessary page faults due to trapping writes to irrelevant data
- It creates an aggregated shadow copy of all protected hooks in a centralized location

Hypervisor

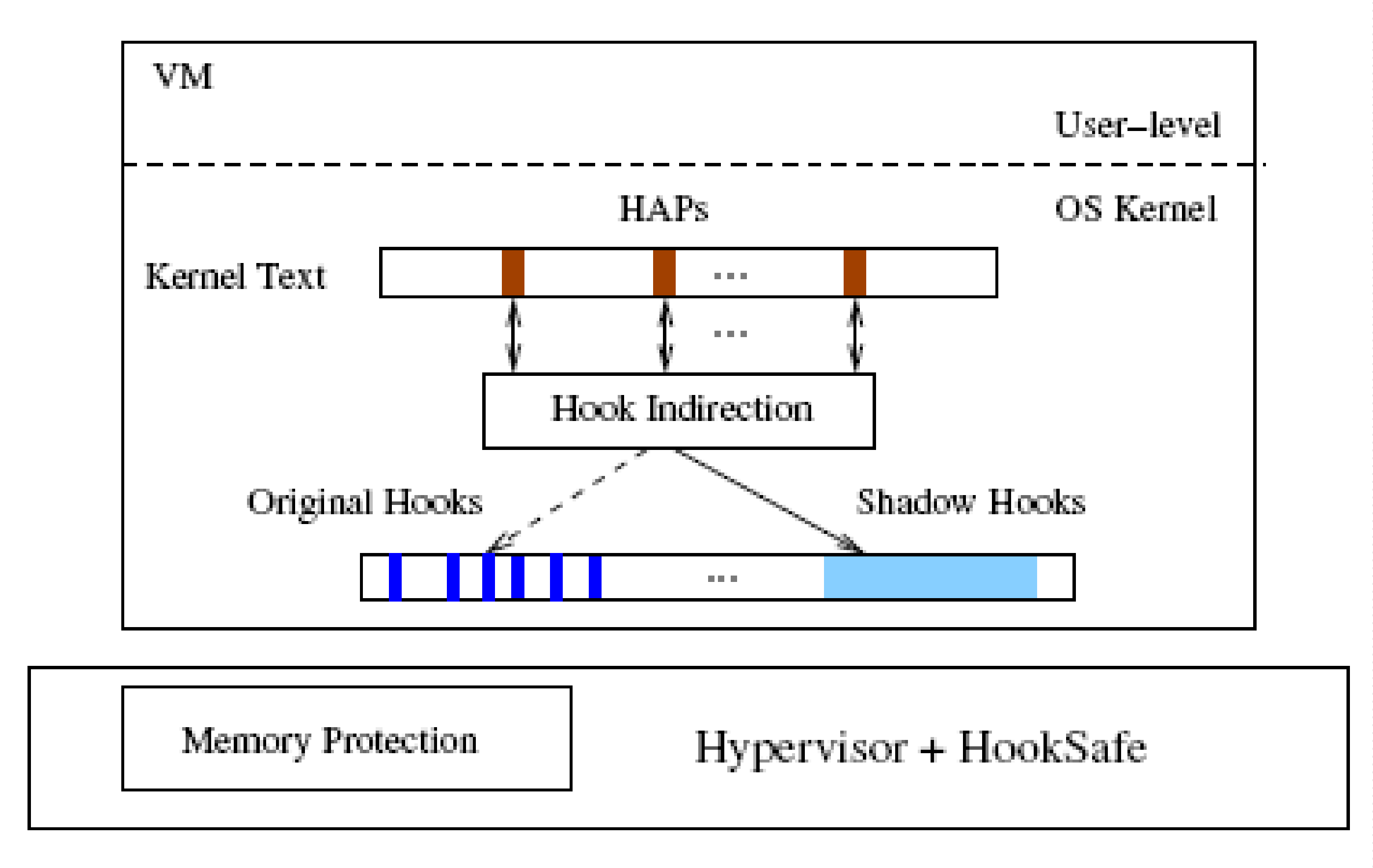
- HookSafe is based on the Xen hypervisor (VM)
  - HookSafe replaces HAP instruction at runtime with jump to trampoline code
  - Trampoline code collects runtime context to determine kernel hook being accessed
  - It redirects read accesses to appropriate shadow hooks
  - Any attempt to modify the shadow copy is trapped and verified by the hypervisor

HookSafe Architecture

Two key steps:

1. Offline hook profiler profiles guest kernel execution and outputs hook access profile for each protected hook
   - Currently based on emulation and monitoring of the target system
2. Online hook protector creates shadow copy of protected hooks and instruments HAP instructions to redirect their accesses to shadow copy

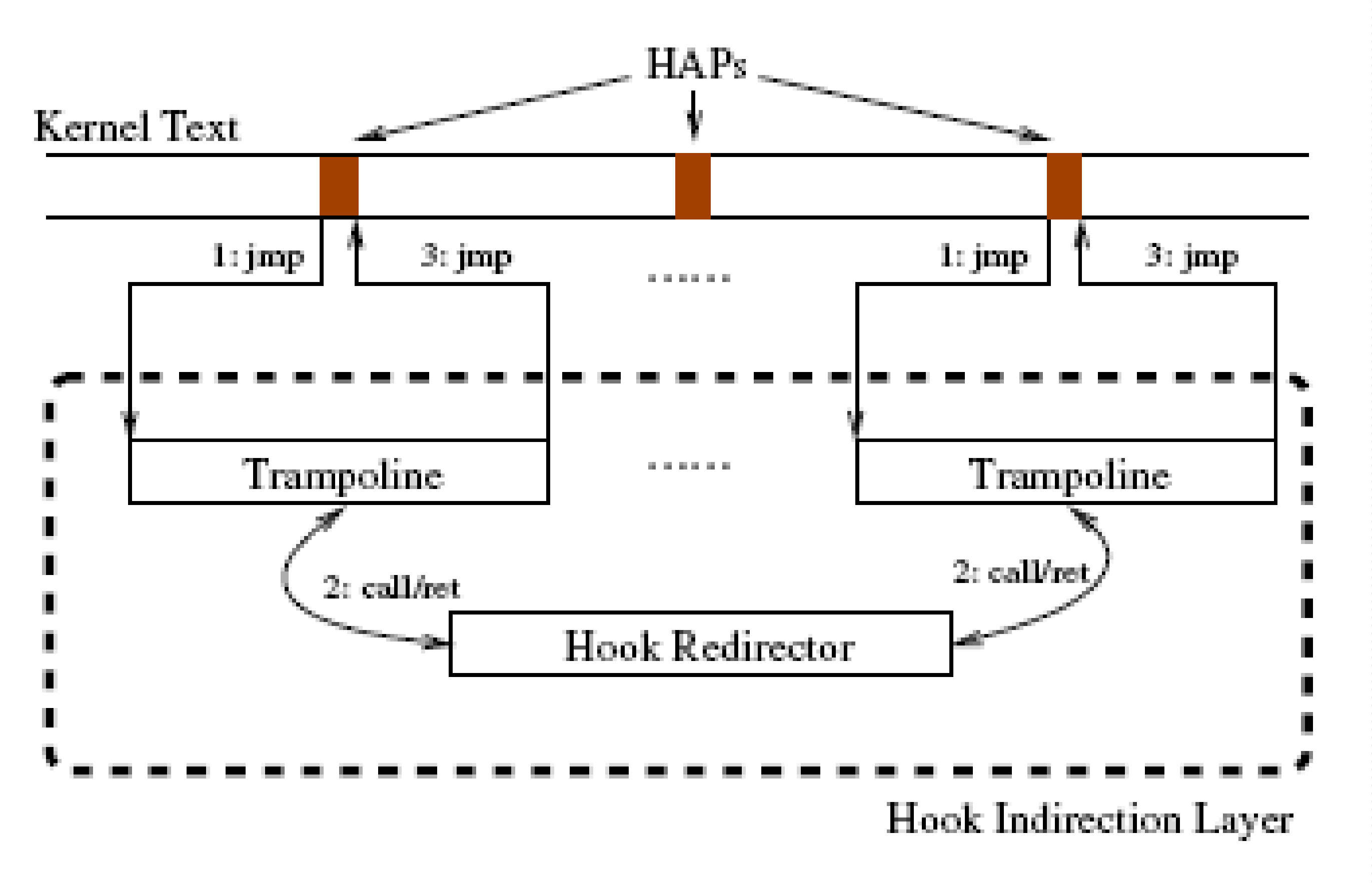
Online Hook Protection



Hook Indirection

- For read accesses, the indirection layer simply reads from the shadow hooks then returns to the HAP site
- For write accesses it issues a hypercall
  - The memory protection component in the hypervisor validates the request and if it is valid updates the shadow hook
    - The new hook must have been seen during profiling
    - The hypervisor replaces the HAP instruction with a *jmp* to trampoline code
- The memory allocation/deallocation functions used by the kernel are instrumented to handle dynamically allocated hooks

Hook Indirection (2)



Limitations of HookSafe

- The hook access profiles may be incomplete
  - May be addressed by incorporating static analysis
- HookSafe assumes prior knowledge of the kernel hooks that should be protected