Detecting Exploit Code Execution in Loadable Kernel Modules

Haizhi Xu Wenliang Du Steve J. Chapin
Systems Assurance Institute
Syracuse University
3-114 CST, 111 College Place, Syracuse, NY 13210, USA
{hxu02, wedu, chapin}@syr.edu

Abstract

*In current extensible monolithic operating systems, loadable kernel modules (LKM) have unrestricted access to all portions of kernel memory and I/O space. As a result, kernel-module exploitation can jeopardize the integrity of the entire system. In this paper, we analyze the threat that comes from the implicit trust relationship between the operating system kernel and loadable kernel modules. We then present a specification-directed access monitoring tool—*HECK*, that detects kernel modules for malicious code execution. Inside the module, HECK prevents code execution on the kernel stack and the data sections; on the boundary, HECK restricts the module's access to only those kernel resources necessary for the module's operation. Our measurements show that our tool incurs 5–23% overhead on some I/O intensive applications using these modules.*

1. Introduction

Loadable Kernel Modules (LKM) are kernel extensions that can be loaded into operating system kernel dynamically. In extensible operating systems, LKMs are part of the kernel and are implicitly trusted once loaded into the kernel. The modules are in the same hardware protection domain as the hosting kernel and they can access the entire kernel address space and execute privileged instructions. Most kernel-space device drivers (e.g., the drivers for sound cards and CDROMs, and many other extended kernel functions such as the DOS FAT file system) are implemented as kernel modules.

According to the study of Chou and his colleagues [5], kernel modules constitute 70% of Linux kernel code; they account for 70% to 90% of well-known kernel bugs; and the average bug life time for kernel modules is 1.8 years. Many kernel modules (e.g. device drivers) are developed under time restrictions and therefore may not be fully tested for security properties. Vulnerabilities such as lack of boundary and pointer checks can lead to kernel-level exploits, which can jeopardize the integrity of the running kernel. Inside the kernel, exploit code has the privilege to intercept system service routines, to modify interrupt handlers, and to overwrite kernel data. In such cases, the behavior of the entire system may become suspect.

Kernel-level protection is different from user space protection. Not every application-level protection mechanism can be applied directly to kernel code, because privileges of the kernel environment is different from that of the user space. For example, non-executable user page [21] and non-executable user stack [29] use virtual memory mapping support for pages and segments, but inside the kernel, a page or segment fault can lead to kernel panic. In addition, the system hardware protection does not isolate kernel modules from the core kernel and each other. The differences show us that protecting kernel modules requires both sandboxing and malicious code prevention.

This paper introduces the design and implementation of *HECK*—Hybrid Extension Checker for Kernels—a specification-directed security monitor for detecting exploit code execution in kernel modules. HECK restricts external access from an instruction in a kernel module to only the legitimate addresses listed in the specification of the module. In addition, HECK prevents the module from executing code on the kernel stack or in other data sections of the module.

The contributions of this paper are

1. analyzing the threats from kernel-module exploitation and the issues in defending kernel-level attacks;

2. detecting malicious code execution from a kernel module using a combination of module isolation, specification based access control, and software-enforced non-executable data sections.

Throughout this paper, we use the terms module, kernel module, and loadable kernel module interchangeably.

2. Problem analysis

In this paper we emphasize protecting benign but vulnerable (e.g. careless boundary checking) modules. Implementation errors in kernel modules may be exploited by non-privileged attackers, as long as they have access to the device using a vulnerable module. For example, the Windows ping-of-death packet exploits a careless calculation on the size of an IP packet, which can lead to smashing the packet buffer and beyond [11]. An error in the ISO9660 file system that performs no checking on the size of symbolic file names can lead to buffer overflow attacks by feeding a malformed CD with long symbolic files [17]. None of the attacks require root privilege.

The threat that allows kernel modules to compromise the underlying operating system kernel is due to the implicit trust relationship between the operating system and its extensions—loadable kernel modules. In monolithic kernels, exploit code in a kernel module can access entire kernel code, data structures, hardware interface, and even the system protection mechanism. As a result, kernel-level exploits can break the integrity of the run-time kernel. For example, exploit code can disable system security mechanisms, break system protection, modify system behavior, escalate user privileges, and deny system services.

Among all the attacks a kernel module faces, malicious code execution is the most powerful approach for attackers and is our main target for protection.

Malicious code execution can be launched in the following ways:

1. transferring control from a kernel module to a piece of exploit code located in the data sections of the module, on the kernel stack, or in a kernel buffer. This is analogous to a shell code attack in the user space;
2. calling an unauthorized kernel function, similar to the return-into-lib(c) attack in user land;
3. overwriting unauthorized kernel data structures containing function pointers, which can be called later on;
4. calling authorized functions with malicious parameters.

All the above attacks need execution of a *sensitive instruction*, either a branch instruction or a data access instruction overwriting a code pointer, which changes the program behavior from benign to malicious.

Sandboxing or isolating a kernel module prevents access to unrelated kernel sections, but it is not enough for protecting kernel modules. First, sandboxing does not provide fine-grained controlled access to the kernel (e.g., allowing only external calls to legitimate kernel functions and returning to legitimate locations), which is required by the functionality of device drivers; second, sandboxing does not prevent code execution on the stack or within other data sections inside the module.

Due to the limitations above, we augment sandboxing with malicious code prevention. Malicious code prevention covers static analysis, compiler-based detections, and run-time monitoring depending on the deployment stage. We choose run-time monitoring among these techniques because it provides the best support for legacy modules. Static analysis and compiler-based techniques may not be feasible if the source code is not available, which is especially true for Windows-based device drivers. Besides, our run-time monitor can be merged with software-based isolation by using code instrumentation techniques.

3. Overview of HECK

We focus on detecting and preventing exploit code execution in vulnerable kernel modules. Our work is based on three assumptions:

- there are unexpected implementation errors in the legitimate kernel modules;
- the attacker has no root privilege, but is authorized to operate on a device using vulnerable kernel modules;
- the source code of the modules are trusted, compiled with a trusted compiler, and the binary images are stored safely. In Linux, the default system configuration requires root privileges to modify the on-disk kernel module images.

3.1. Restrictions on the kernel modules

From the analysis in section 2 we know that the sensitive branch instructions and memory access instructions are critical to exploit code execution and therefore should be carefully monitored. We isolate the module, except for allowing selective operations based on a specification of the legitimate requests. We perform the following restrictions on the control flow and data flow instructions of the module being monitored:

1. We monitor branch instructions, if the target addresses can be derived at run time. Inside the module, we allow control transfer within the code section and disallow control transfer to the data sections, heap, and the kernel stack. The target address, if out of the module, must be explicitly named as a legal branch target by the specification.

   Inside the module, malicious code can be located in the data sections or on the kernel stack, or in the middle of a legitimate instruction (this attack is not applicable to RISC machines because the size of a RISC instruction is fixed). We mark the data sections and the kernel stack as non-executable sections and check on

every sensitive branch instruction for violation. To prevent jumping to the middle of an instruction, we can add inter-instruction padding randomly, thereby moving potential target instructions to unpredictable offsets from their original locations. This makes the attack likely to crash rather than finding desirable locations.

Outside the module, a potentially malicious target address can be at any location in the kernel. We extract the legitimate external calls to the kernel functions from static analysis. Then we monitor the external calls to ensure that execution passes through only these legitimate entrance locations and return to the corresponding legitimate return addresses.

2. In order to avoid unauthorized writes to the unrelated kernel data, especially code pointers, we do not give a module unnecessary permissions access to the kernel data.

   To prevent caller frame pointers and return addresses from being overwritten, we disallow write accesses to the stack outside the local frame.

   Using static analysis, we can obtain only a subset of the legitimate kernel data accesses. Others are obtained through function call parameters and resource allocation functions such as `kmalloc()`. We trust the kernel structures introduced by the permitted kernel functions. If a kernel function calls a module function, we allow the module function to access to the parameters from the caller and treat hierarchical references the same way. We allow access to dynamically allocated resources created by the module.

Our security check works on the instruction level. On this level we do not have full semantic understanding of the program, so data flow monitoring is difficult. Nevertheless, with the type information from the kernel header files, we check on some functions that can overwrite arbitrary memory. For example, we intercept `strcpy()` to check whether the source address and the target address violates the specification.

3.2. The architecture of HECK

One of our goals in designing HECK is providing security monitoring that is compatible with the legacy kernel module code. HECK does not require changing the architecture of the kernel nor does it require module source code. Instead, HECK deals with compiled modules directly.

HECK can be divided into a specification generation part and a run-time monitoring part. Statically, HECK generates a specification describing the module's legitimate operations on system resources (i.e. kernel functions, kernel data structures, and I/O ports). The initial static specification is augmented with information describing dynamically

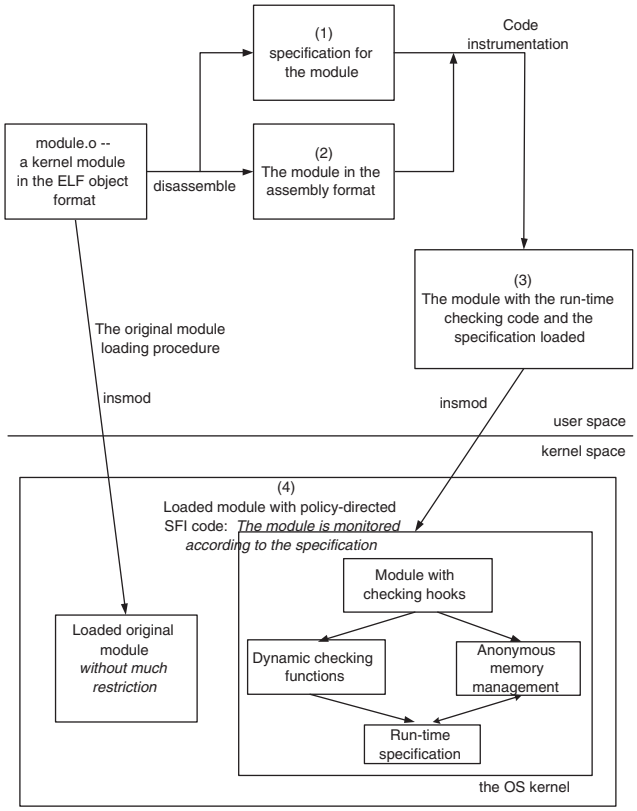

Figure 1. The procedure of monitoring a kernel module. The left side path shows the current module loading procedure. The right side path describes the steps of the HECKing procedure.

allocated resources and parameters-introduced kernel structures at runtime. After generating the static specification, HECK modifies the module to incorporate the specification, the run-time checking code, and dynamic resource management code using code instrumentation techniques. At runtime, HECK monitors the behavior of the module based on the specification.

The following steps describe the procedure of HECKing a kernel module in a temporal order, which is also illustrated in figure 1:

1. Generate a specification of the binary module using `objdump`, a GNU disassembler. This step will be further explained in section 4.1.

2. Disassemble the kernel module.
   In Linux, a compiled module is an OBJ file in the ELF format. It needs to be linked at module loading time to resolve the external symbols (e.g. names of kernel

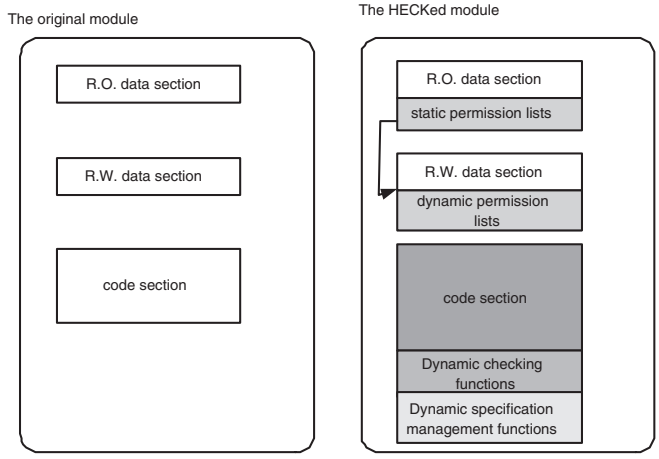

Figure 2. The organization of an original module file and its HECKed version. The permission lists are appended to the read–only section and the read–writable section. The code is instrumented for the dynamic checking and dynamic resource management procedures, which will be further illustrated in figure 3 and section 4.2. The main body of the dynamic checking code and the dynamic resource registration code are appended to the code section.

functions and data structures). An OBJ file contains enough information to restore the assembly file.

We use `objdump` for disassembling. The output of `objdump` contains all the information of the assembly code but not a complete assembly file. We have written a script to convert the output of `objdump` into the assembly format file.

3. Augment the module with the specification, run-time checking code, and dynamic resource management code. The assembly code, including the run-time checking subroutine, checking hooks, and the specification, is re-assembled. Hooks for registering return addresses and parameters are set at the entrance of related module functions. Hooks for registering the dynamic buffers are set after returning from the resource allocation functions.
4. Perform run-time security checking at every check point marked in the above phase. Dynamic resources are registered by the activation of proper registration functions at run time.

Figure 2 illustrates the organization of an original module and its HECKed version. Permission lists derived from the specification are stored at the end of the read-only and read-writable sections. The run-time checking routines and dynamic specification management routines are attached to the code section. The code section is modified for triggering the dynamic checking functions and dynamic resource management functions at the check points.

4. Implementation of HECK

4.1. Description of the specification

A specification describes permissions granted to a kernel module. We represent a *permission* as an operation on a block of virtual memory or I/O ports, marked by the starting address and the ending address. *Operations* on kernel symbols and kernel memory are *read, write,* and *call*. *Operations* on I/O ports are *read* and *write*. The specification contains five pairs of permission lists: *allow_mem_read* and *neverallow_mem_read*, *allow_mem_write* and *neverallow_mem_write*, *allow_mem_call* and *neverallow_mem_call*, *allow_IO_read* and *neverallow_IO_read*, and *allow_IO_write* and *neverallow_IO_write*. The *allow* lists, as the name implies, define the permissions granted to the module. The *neverallow* lists define operations prohibited to the module. At each checking point, we choose one pair of allow/neverallow list, according to the operation and resource type.

As we mentioned in section 3.2, a specification contains a statically defined part and a dynamic part—a dynamic resource manager. The static specification describes accesses to static resources, e.g. kernel functions or kernel data, the addresses of which can be determined statically from the system map. The dynamic part lists accesses to dynamic resources obtained at run time.

The following shows a fraction of the static specification generated for *ide-cd*, the Linux IDE CDROM driver.

```
# default: neverallow stack call
# default: neverallow data_sections call
allow kmalloc_Rsmp_93d4cfe6 call
allow kfree_Rsmp_037a0cba call
allow printk_Rsmp_1b7d4074 call
allow sprintf_Rsmp_1d26aa98 call
allow atapi_output_bytes_Rsmp_affe3a66 call
... ...

# IDE 1
allow IO_port 0x170--0x177 read-write
allow IO_port 0x376--0x376 read-write
# Bus master IDE
allow IO_port 0xd000--0xd00f read-write
neverallow proc_root_Rsmp_020bc977 write
```

Readers should keep the following in mind:

- The *neverallow* permissions supersede the *allow* permissions, if there is an overlap of memory or I/O blocks on both the lists.
- Write permission does not imply read permission by default. This is to maintain consistency with I/O requests, because an I/O port can be read only, write only, or both readable and writable.

We have built *specgen*—a static analysis tool for generating specifications. `Specgen` generates necessary permissions for a legitimate binary module. Using the output from `objdump`, `specgen` lists all external accesses with unresolved kernel symbols and absolute addresses. Since we assume the module is benign, we take every external access at this stage as legitimate. Then `specgen` converts these accesses into permissions in the specification.

Dynamic resources, such as function call parameters, dynamically allocated memory, dynamically memory mapped I/O, and function return addresses, can complicate the monitoring because the location of these parameters are not available statically. We will further explain the procedure of dynamic resource management in section 4.2.

4.2. Implementation of the checking procedure and dynamic resource management

HECK run-time checking, as illustrated in figure 3, is specification directed. The whole checking procedure can be divided into three steps.

Step 1, statically, HECK injects a checking hook before every sensitive instruction. Such a sensitive instruction can be a memory access instruction, a control transfer instruction, or an I/O instruction. For example, in figure 3, before the `ret` marked as `original instruction 12`, HECK adds a checking hook of 4 instructions.

Step 2, before executing the sensitive instruction, a checking hook will call the corresponding monitoring subroutine for checking whether the sensitive instruction conforms to the security specification. For each sensitive instruction in the module, HECK first extracts the access request and searches the request in the appropriate *neverallow.op* list. If there is a match, a violation-handler function is called. If there is no match in the *neverallow* list, HECK searches in the corresponding *allow* static and dynamic lists. In the above example, the program triggers the checking routine `d_policy_fret`, which examines the target address. First, if the target address is located on the kernel stack or a data section, the instruction has a violation; second, if it is in the code section or an allowed kernel entrance point, the instruction is allowed; otherwise, the instruction violates the specification.

Step 3, if the checking code finds a violation, it calls a handler function. Currently, the handler prints a warning message to the system log and returns; it could take more sophisticated action such as unloading the kernel module, although this has potentially dire side effects.

The dynamic resource management works similar to the dynamic checking procedure. First, statically, we have inserted hooks for dynamic registration functions at the entrance of the cross-boundary functions in the module and after kernel resource allocation functions.

Then, at run time, the hook triggers a function for registering either the return address or dynamic resources. By the time the registration function is triggered, the resource pointers already exist on the stack. The registration function will put the locations of the dynamic resources into the dynamic permission lists. HECK registers a dynamic buffer after returning from dynamic memory allocation functions, such as `kmalloc()` and `ioremap()`; HECK also registers the return address and the function call parameters at the entrance of the called function, if the caller is external to the module. The hierarchical structures in the parameters are defined in the kernel header files. We interpret the hierarchy manually and hard code it into the registration functions.

For example, in figure 3, at the entrance of the function `cdrom_log_sense`, a dynamic resource management function `d_policy_fenter` puts the return address to the `allow_mem_call` list.

4.3. Scope of protection

HECK is designed to prevent exploit code execution from inside kernel modules. Technically, HECK combines kernel module isolation, access control, and prevention of malicious code execution.

The specification should be kept away from any unauthorized manipulation. As we can see in figure 2, we store the permission lists of the static specification in the read-only data section, to prevent them from being overwritten. The dynamic permission lists are located at the end of the regular data section. To prevent unexpected overwriting of permissions in the writable section, we put the subsection holding the permission lists into the *neverallow_mem_write* list, which is located in the read-only section.

Complete data flow checking requires semantic-level understanding of the program. Although HECK checks on memory access instructions in the module, as a binary-level monitoring tool, HECK is incapable of performing complete data flow protection. What HECK can protect is directly overwriting unrelated kernel structures such as the system call table. In the kernel modules we studied, we observed that a kernel module accesses a limited range of kernel data, which is amenable to monitoring. For example, the IDE cdrom driver only uses $hwif \rightarrow drives[i]$ and its hierarchically related structures.

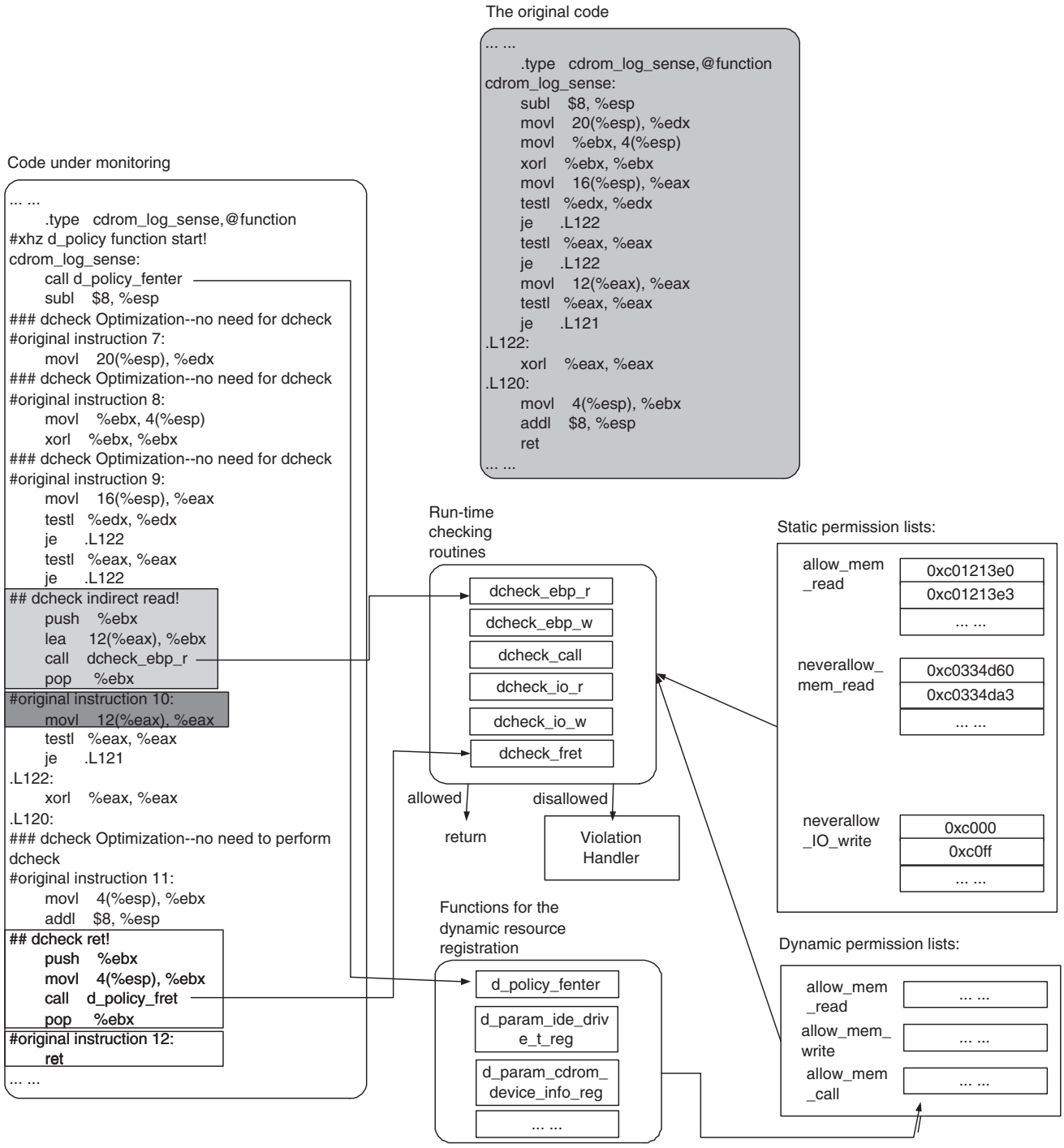

Figure 3. HECK run-time checking and dynamic resource registration for the IDE CDROM driver. The shaded code on top is part of the original code. The unshaded code block to its left is the corresponding HECKed code, in which the instruction to be examined is marked by dark shade; and a hook of the corresponding checking function above it is marked in lighter gray shade. Function `d_policy_fenter` is triggered at the entrance of the function to register the return address to the dynamic permission lists. `dcheck_fret` is called to examine the return address. `dcheck_ebp_r` is used to check the memory read instruction.

Because data flow intermingles with control flow (e.g. a code pointer is a piece of datum), HECK, or other instruction level checking tools, cannot eliminate control flow attacks. The malicious code may call a legitimate but vulnerable kernel function (e.g. `printf()` with the `%n` vulnerability) to overwrite a code pointer in the kernel to point at a piece of attacking code, which can be triggered latter. HECK is not designed to make up for the vulnerabilities in the kernel; instead, HECK relies on the correct behavior of the kernel functions. An ad hoc solution for this problem is to analyze statically or profile the legitimate parameters for external calls and to use the information in intrusion detection. Nevertheless, HECK prevents common attacks such as exploit code execution on stack (or in other data sections) and manipulating non-related kernel resources (e.g. redirecting the system call table).

5. Overhead measurement

We have measured the overhead of HECK on a Pentium II 400MHZ dual-processor SMP PC running Red Hat Linux 8.0 with a 40Xmax ATAPI CDROM drive and a generic 10/100M PCI network interface card. The kernel version number is 2.4.18-14smp.

In our experiments, we chose `cdrom`, `ide-cd`, `8139too`, `fat`, and `ext3` as demonstration kernel modules. `Cdrom` is a Linux high-level uniform CD-ROM driver. `Ide-cd` is a Linux ATAPI CD-ROM driver. `8139too` is a Linux fast Ethernet driver for boards based on the RTL8129 and RTL8139 PCI Ethernet chips. `Fat` is the DOS FAT file system module, and `ext3` is the Linux EXT3 file system module. We store the specification in files with `.specification` appended to the name of the module. A fraction of the specification for `ide-cd` can be found in section 4.1.

While we have not conducted performance measurements on the portion of the kernel module that is performing low-level interrupt handling (and is therefore time critical) directly, our analysis of the IDE-CD device driver and network driver shows that our mechanism will not adversely affect bottom-half interrupt handlers. The bottom-half interrupt handler is structured to do very little work before returning, so there are few control transfers that we need to check. We have run the HECK version of these device drivers on a desktop workstation that experiences daily use over a multi-month period with no ill effects.

We measured the space overhead of the run-time checking code. Table 1 lists the number of run-time checking points we put in the example modules. Certain sensitive instructions, such as load and store instructions manipulating local variables and PC-relative addressing mode jumps, do not need run time monitoring, because we can find the relative target locations statically. On average, 7% of the instructions that we monitored need run-time checking.

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

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.

Figure 4 shows our measurement of the overhead of the run-time checking on a CPU intensive micro-benchmark program. This serves as an upper bound on the runtime overhead of our checking code. Real I/O-bound device drivers do not call checking routines so frequently, because the driver is idle when no I/O requests are being processed. With 20% of the instructions being checked, the overhead of the run-time checking is about 12 times the original running time; with 15% of the instructions being checked, the overhead of the run-time checking introduce 8.5 times overhead. In short, the HECKed version module runs about 10 times slower than the original one in the worst case.

We measured the overhead of run-time checking code on the Linux CDROM driver, network interface card driver, and FAT and EXT3 file system modules. Figure 5 shows the overhead of the run-time checking on some practical I/O intensive programs. We measured 1%–3% overhead of our dynamic monitoring on these I/O intensive applications with a *allowall* specifications, and 5%–23% overhead with the realistic specifications. In the *allowall* specification, we put the whole kernel segment as the first entry of the *allow* lists and leave the *neverallow* lists empty. It has the minimum overhead for our checking procedure because the code does not need to go deep into an *allow* list for a match. With a more realistic specification, the checking function needs to go through the specification lists for a match.

We find that the more I/O intensive a module is, the less overhead our dynamic monitor puts on the module. This observation is reasonable. In an I/O intensive program, a

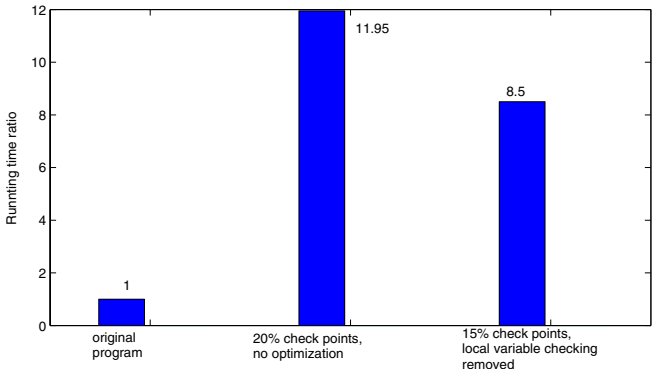

Figure 4. Run-time overhead on a CPU intensive application micro-benchmark, serving as the upper bound for the device drivers

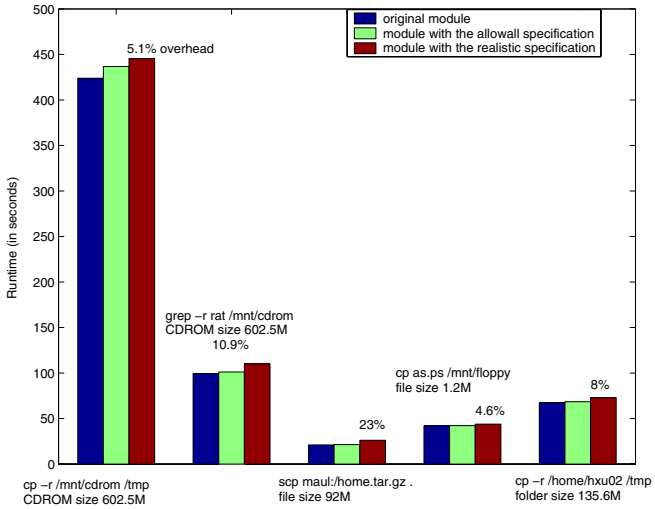

Figure 5. Overhead of our dynamic monitoring on some I/O intensive application programs

large proportion of the real time is spent on I/O operations, while both the driver module and the application program are sleeping. The overhead of dynamic monitoring on the driver is limited by the proportion of the module running time to the overall running time. We measured (4.6%–8%) overhead on the three `cp` programs with realistic specifications. The overhead on the FAT floppy drive file system (4.6%) is smaller than the overhead on the CDROM driver (5.1%) and on the EXT3 file system (8%). Because floppy disks are much slower than hard disk and CDROM, the FAT file system is more I/O intensive than CDROM and EXT3, and therefore, less affected by our monitoring.

The overhead on `scp` over network (23%) is significantly higher than the other four measurements. We consider the following reasons for the overhead variation: First, the network driver operates on smaller blocks than the local CDROM driver and EXT3 file system. Second, because file systems can perform prefetch that networks cannot, the network driver is called more frequently than the CDROM driver and EXT3 code for the same amount of data. Third, the network driver has 16% of its instructions being checked dynamically, compared to 6% on EXT3, and 16% on the CDROM driver. Finally, the specification of the network driver contains longer permission lists than the CDROM driver and the EXT3 module. So far, we use a linear search on the lists. A binary search on the static permission lists should reduce the per-checking cost.

6. Related work

Related work includes kernel module isolation, user space malicious code detection, and policy directed programs safety and security checking.

Prior work on kernel modules has focused on in-kernel programs isolation [3, 22, 20, 25, 27, 30]. There are three techniques to isolate a kernel module: one is to put the modules in a separate hardware protection layer (e.g. ring 1 or 2 [4], or the user level [8, 26] in the Intel x86 architecture); the second is to use the segmentation hardware in ring 0—the kernel protection layer [30]; and the third is to apply a per-instruction boundary checking inside the same hardware protection domain as the kernel [25, 27].

Software fault isolation (SFI) [27, 34], or sandboxing, is effective in isolating in-kernel applications. SFI emulates hardware protection by checking each control-transfer instruction to prevent the isolated program from escaping from its domain. Existing SFI tools are segmentation-based, and they work well for isolating loosely-coupled programs, which access only their own data and code, and calling the system service routines through only the system call interface or IPC interface. Addresses in such programs can be identified by the segment tags. Swift and his colleagues isolate kernel modules using segmentation hardware [30].

Kernel module isolation did not take security into full consideration. Under this frame work, attacks on a vulnerable kernel module can subvert the system, for example, by jumping to the exploit code on the stack. Our work improves the SFI technique by incorporating access control and malicious code prevention. We chose SFI-based isolation instead of the others because it can incorporate malicious code detection conveniently through code instrumentation.

Kernel modules in the SPIN operating system are writ-

ten in a type safe language [3]. This prevents many implementation errors but does not support legacy binary modules that are written in other languages.

In the user space, much security research has been done in preventing malicious code from execution. Stackguard and pointguard are effective compiler-based security techniques in preventing malicious code execution. Stack guard puts canary words before the return address to detect stack smashing attacks [7]. Pointguard encrypts legitimate pointers to detect malicious pointers [6]. Type assisted overflow detection [15] compiles type information into the binary code and detects type errors at run time. These compiler-base techniques require source code, while our HECK works on the binary code directly.

HECK is also influenced by specification-based IDS [23], policy directed code safety and security [9, 10, 16, 19, 24, 28, 32], static analysis [1, 31, 33], and dynamic code monitoring [12, 27, 32]. There are two techniques suitable for dynamic code monitoring: code interpretation [32] and code instrumentation [27]. We choose code instrumentation in our implementation because it is convenient to deploy in the kernel.

Although there are various solutions proposed dealing with problems related to kernel modules, a satisfactory solution is still yet to come. Monitoring critical kernel data structures but leaving other parts unprotected is subjective to adaptive attacks [13, 14]. LIDS disables LKM functionality after boot-up [35], which reduces the convenience of using LKMs. Rootkits detection [18] is signature based malicious module detector, which is difficult to catch unknown attacks. Loading modules with only registered signatures [2] can authenticate the modules, but does not guarantee that the modules do not contain any vulnerabilities.

7. Conclusion

Kernel module exploitation can jeopardize the integrity of operating systems. It requires permissions of access to a device using vulnerable kernel modules, which permission is usually granted to ordinary users. This issue has been overlooked by the academic research community, although much related work studies kernel module isolation and user space exploit detection.

In this paper, we propose an approach to detect in-module exploit code execution on the instruction level with the help of a specification. HECK, the tool we developed, has the following features:

- HECK combines module isolation, specification based access control, and exploit code prevention.
- HECK uses code instrumentation technique to monitor the module at run time. The monitoring of a module relies on a specification, which describes the legitimate requests of the module.
- HECK works on the binary code level, without requiring the source code.

As an instruction-level technique, our tool can examine every branch instruction inside the kernel modules. On the other hand, tools at this level do not monitor the data flow completely, because data flow monitoring requires semantic understanding of the program behavior. In data flow monitoring, we prevent only direct access to unrelated kernel data sections. Our future work is to study the data flow properties in a semantically sensitive way and examine how control flow and data flow are interrelated with each other in kernel modules. The data flow pattern of the kernel modules in our study shows that a device driver visits limited kernel structures hierarchically through a pointer parameter. Many important kernel structures (e.g. the system call table) that attackers widely exploit are not necessary for the legitimate operations. This property can simplify the data flow monitoring for kernel modules.

We have implemented HECK for LKMs on an x86 Redhat Linux PC. We have run HECK-modified kernel modules continuously on a desktop workstation for a few months with no ill effects. Our analysis and experiments show that our tool can detect a large set of malicious code execution from within kernel modules. We measured 5%–23% overhead on some I/O intensive applications because of our dynamic monitoring on the kernel modules that we tested.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of 2002 IEEE symposium on security and privacy*, 2002.

[2] M. Bernaschi. REMUS: a security-enhanced operating system. *ACM Transactions on Information and System security*, 5(1):36–61, February 2002.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G.Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems principles*, pages 267–284, 1995.

[4] T. Chiueh, G. Venkitachalam, and P. Pradhan. Intra-address space protection using segmentation hardware. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, March 1999.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, October 2001.

[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[8] J. Elson. FUSD— a linux framework for user-space devices. http://www.circlemud.org/ jelson/software/fusd/, 2003.

[9] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 New Security Paradigm Workshop*, September 1999.

[10] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of 1999 IEEE symposium on security and privacy*, May 1999.

[11] Fyodor. Ping of death. http://www.insecure.org/sploits/ping-o-death.html, 1997.

[12] L. Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation, Second Edition*. Addison Wesley, June 2003.

[13] K. J. Jones. Loadable kernel modules. *;login:*, pages 43–49, November 2001.

[14] T. Lawless. Saint jude, the model. http://sourceforge.net/projects/stjude, 2000.

[15] K. Lhee and S. J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, August 2002.

[16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[17] G. MacManus. Buffer overflow in iso9660 file system component of linux kernel. *http://www.idefense.com/application/poi/display?id=101 &type=vulnerabilities&flashstatus=true*, April 2004.

[18] N. Murilo and K. Steding-Jessen. Locally checks for signs of a rootkit. http://www.chkrootkit.org/.

[19] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan 1997.

[20] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, October 1996.

[21] W. Purczynski. kNoX—implementation of non-executable page protection mechanism. http://www.opennet.ru/prog/info/1769.shtml, May 2003.

[22] G. Rosu and N. Segerlind. Proofs on safety for untrusted code. *UCSD technical report CS1999-0633*, October 1999.

[23] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.

[24] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model–carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[25] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[26] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *17th ACM Symposium on Operating Systems principles*, Dec. 1999.

[27] C. Small. *Building an Extensible Operating System*. PhD thesis, Harvard University, Division of Engineering and Applied Sciences, October 1998.

[28] S. Smalley. Configuring the selinux policy. Technical report, National security agency, http://www.nsa.gov/selinux/papers/policy2/t1.html, January 2003.

[29] Solar Designer. Non-Executable User Stack. http://www.openwall.com/linux/.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[31] B. v. Chess. Improving computer security using extended static checking. In *Proceedings of 2002 IEEE symposium on security and privacy*, 2002.

[32] S. A. Vladimir Kiriansky, Derek Bruening. Secure execution via program shepherding. In *Proceedings of the 11th USENIX security symposium*, August 2002.

[33] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of 2001 IEEE symposium on security and privacy*, 2001.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993.

[35] H. Xie. LIDS hacking HOWTO. http://www.lids.org/lids-howto/lids-hacking-howto.html, 2000.