# Sentry: a Binary-Level Interposition Mechanism for Trusted Kernel Extension

Se-Won Kim*, Jae-Hyun Hwang*, Jin-Hee Choi**, and Chuck Yoo*
*Department of Computer Science and Engineering
Korea University Seoul Korea
**Samsung Electronics, Suwon, Korea
{swkim, jhhwang, jhchoi, hxy}@os.korea.ac.kr

## Abstract

*Several commodity operating systems have used kernel extensions to extend or replace their functionalities. Generally, since the kernel extensions are executed in the same address space with the kernel, a mere fault in the extensions may lead the whole system to be corrupted. So naturally, studies on the kernel extension are mainly proposed with the goal of isolating extension faults from the system. However, previous schemes require the static analysis of the extension module and the modification of kernel source code. The goal of this paper is to remove such overhead stages.*

*This paper proposes Sentry; a lightweight kernel subsystem that provides dependable execution environment for the kernel extensions. We show the efficiency of Sentry through practical implementation on Linux.*

## 1. Introduction

Contemporary operating systems have used kernel extensions to extend their functionality. A typical example of the kernel extensions is a device driver. Through loading the device driver into kernel address space, the kernel can use a new hardware. As we can see in case of the device driver, the kernel extension makes the kernel flexible and extensible. However, such extension often leads the kernel to unreliable state [7] [9]. For instance, if the extension has fault like accessing invalid address, the whole system may be crashed. Unfortunately it is not easy to protect the kernel from the extension fault because most of modern operating systems do not provide any innerkernel protection. Moreover, short development schedule aggravates the situation.

The interposition service is a well-known approach to construct restricted environment that monitors and emulates the actions kernel extensions take.[13] Before loading a kernel extension, the service inserts hooks, which intercept control flows, into every place where a kernel function is called. And then, the inserted hooks will call a wrapping stub corresponding to the kernel function that the extension calls originally. The role of the wrapping function stub is to interpose kernel extension action. This idea is simple and institutive, but the approach has two drawbacks: 1) the interposition services require modification of the kernel and the extension source codes. 2) The interposition services should be rewritten and recompiled when the kernel needs a new wrapping stub. Providing the interposition service in binary level, such drawbacks would be removed, and eventually we can provide more efficient and trusted environment for the kernel extension.

This paper proposes the architecture and design of Sentry, a lightweight kernel subsystem that provides trusted execution environment for the kernel extensions. Most notable thing is that our scheme operates in not source code but binary level. Sentry only parses ELF object file, and it appends information for hooking the functions to a temporary object file. Sentry loads the temporary file into memory, and finally the functions in the extension image are monitored and regulated by our mechanism.

The remainder of this paper is organized into following sections. Section 2 reviews related work that tries to solve unreliability of the kernel extensions. Section 3 describes goal and design of Sentry. In Section 4, we present how describe policy issues of Sentry and implementation of policy. Finally, Section 5 summarizes out work and draws conclusions.

This work was supported by grant No.R01-2004-000-10588-0 from the Basic Research Program of the Korea Science & Engineering Foundation.


0-7695-2687-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

## 2. Related Work

There are several projects to improve reliability and dependability of operating systems. We classify them into four types:1) approaches based on hardware, 2) approaches based on software, 3) user-level kernel extension and 4) type-safe programming language and compiler assist.

Hardware based approaches can be divided into two category. First is that kernel extensions execute in different hardware protection level like ring 1 or 2 in the Intel x86 architecture. [10] proposed execution of kernel extension in ring 1 while kernel runs in ring 0 and user level processes run in ring 3. Second is using segmentation hardware in ring 0. Nooks[8] executes kernel extensions and kernel in same protection level but they has different access right of each other's address space. It permits kernel extensions read/write access rights in their own address space, but only read right in the rest of kernel address space. [10] must differently configure segmentation hardware which uses more than two protection levels. Nooks modifies the standard kernel module loader and the kernel's module initialization code. Since these mechanisms use specialized paging mechanism and kernel subsystem, modification of kernel source code should be required.

HECK[6] and Software Based Fault Isolation(SFI)[2] are software based approaches. Both projects propose instruction checking method. They define memory access instruction, control transfer instruction, and I/O instruction as sensitive instruction. After scanning and analyzing assembly code of kernel extension, and then they insert a hooking code in front of sensitive instructions. An inserted hooking code moves flow of execution to run-time system of kernel and it decides whether permit the instruction or not. However, finding sensitive instruction and inserting hooks in binary code are difficult and complex process.

[3],[4] and [11] put kernel extensions in user-level. Kernel extension and kernel communicate with each other using IPCs. Since any fault in user-level can be protected from kernel and other user-level tasks, putting kernel extension in user-level address space improves reliability and dependability of operating system. Because communication between kernel and kernel extension are very frequent and expensive, however, this approach should endure performance degradation.

## 3. Goals and Design of Sentry

This section describes goals and design of Sentry. As depicted in Figure 1, Sentry consists of two parts: Sentry run-time system and kernel extension pre-processor. Sentry run-time system, as Loadable Kernel Module (LKM), executes in kernel address space and maintains wrapping stubs which interpose kernel extension actions. Kernel extension pre-processor, as user mode process, modifies kernel extensions at binary code level to hook actions of kernel extension.

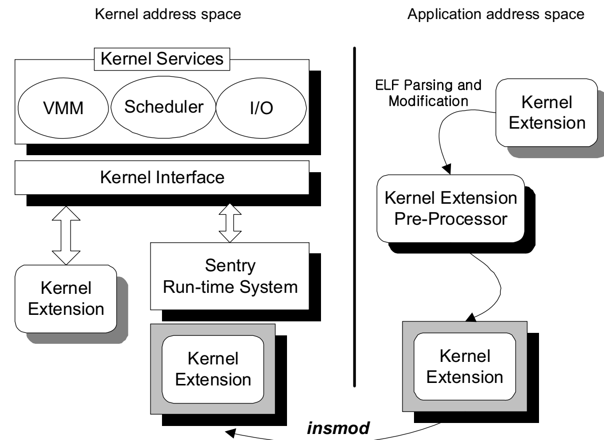


**Figure 1. Architecture of Sentry**

### 3.1. Goals

Sentry has three goals.

1. Monitoring kernel extensions behavior: Sentry monitors what kernel routine is called and what kind of kernel resource is used by kernel extensions. Through monitoring kernel extensions' behavior, operating system can prevent whole kernel corruption like kernel panic. Moreover, such information would be helpful to kernel extension developer.

2. Compatibility: Sentry can cooperate with existing Linux kernel and existing kernel extensions. Compatibility is considered as very important factor when we design Sentry. Sentry supports two common conventions of Linux kernel; the ELF file format [5] and the C programming language. The ELF has become the most widely used binary format in Unix-like operating systems. The C programming language, also, is the most preferred system programming language.

3. Flexible kernel run-time system: To provide interposition service, kernel run-time system keeps wrapping stubs. Projects like Nooks and HECK have kernel run-time system that includes wrapping stubs statically, and such wrapping stubs are various including virtual memory management, file system, network protocol, and hardware I/O. To replace or add new wrapping stubs, however, monolithic run-time system must be recompiled and reloaded into kernel address space.


IEEE COMPUTER SOCIETY

To design Sentry as flexible structures, we select two design schemes: 1) separating wrapping stubs from run-time system and 2) providing programming library to help implementing wrapping stubs. These two schemes are described in following subsections.

## 3.2. Design

### 3.2.1. Sentry Run-time System

Sentry run-time system (Figure 2.) consists of wrapping stubs, run-time system engine, policy, and resource manager. Wrapping stubs is a collection of functions that are called by modified kernel extensions. Since all wrapping stubs are not built-in Sentry, we can compile wrapping stubs and the rest of Sentry separately. They are implemented as a Loadable Kernel Module (LKM)[1], and with help of Linux kernel module loader, wrapping stub can load into kernel address space at any time after loading Sentry run-time system.

Role of run-time system engine is to manage wrapping stubs and generate log. To manage wrapping stubs, run-time system engine provides two interfaces: *regis-ter_wstub()* and *unregister_wstub()*. Each interface is called at *init_module()* and *cleanup_module()* respectively. Because a LKM of Linux must implement such two common interfaces, when we load or unload wrapping stubs, *register_wstub()* and *unregister_wstub()* are called automatically by Linux kernel module loader.

Resource manager tracks system resources which are requested and used by kernel extensions. Besides, it maintains statistics of system resource. We believe that this information will be helpful not only at run-time of kernel extensions, but also at development time of kernel extensions.

Policy specifies kernel routines and system resources permitted to kernel extension. Wrapping stub queries whether requested kernel routine is allowable to kernel extensions through library routines: *query_permission()*. In section 4, description of policy will be presented in detail.

To help implementing the wrapping stub, Sentry provides some library routines. The following describes each function.

- `resister_wstub()`:
  notifies wrapping stub to run-time system engine and checks duplicated wrapping stub for same kernel routine.
- `unresister_wstub()`:
  deletes wrapping stub from Sentry.
- `generate_log()`:
  hands over log message to run-time system engine.
- `link_resource()`:
  register a newly allocated resource. Wrapping stub corresponding to resource allocation routines like kmallc should call link_resource to register type and size of resource in Resource Manager.
- `unlink_resource()`:
  should be called when an allocated resource is freed. It checks that target resource is shared by other component and that kernel extension has right to free target resource.
- `query_permission()`:
  decides that requested kernel routines or kernel resources are allowed to kernel extension.

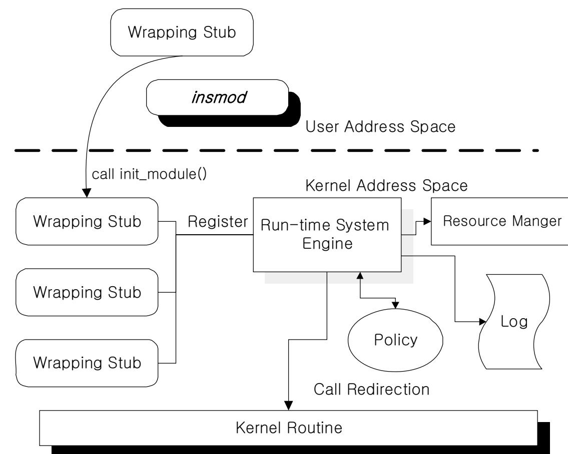


**Figure 2. Sentry Run-time System**

Figure 3 shows an example of wrapping stub of *al-loc_skb()* which allocates *struct sk_buff*. *sen-try_alloc_skb()* will be called by kernel extension instead of *alloc_skb()*. After *sentry_alloc_skb()* calls *alloc_skb()*, it checks policy to decide whether requested operation is allowed or not. The next functions: *init_module()* and *cleanup_module()* call *register_wstub()* and *unregister_wstub()* respectively.

```
stuct sk_buff *sentry_alloc_skb
    (unsigned int size, int gfp_mask)
{
      struct sk_buff ret_val;
      /*Wrapping Routines*/
      ret_val = alloc_skb(size, gfp_mask);

      if( query_permission() )
      {
            /*if operation allowed*/
            link_resource(ret_val);
            return ret_val;
      }
      else
      {
            /*else operation is banned*/
      }
}


int init_module(void)
{
       int retval;
       /*initialization code*/
       retval = register_wstub(sentry_alloc_skb);
       /*test return value*/
}

void cleanup_module(void)
{
       int retval;
       /*cleanup code*/
       retval = unregister_wstub(sentry_alloc_skb);
       /*test return value*/
}
```

**Figure 3. Example of wrapping stub code**

#### 3.2.2. Extension Pre-Processor

Figure 4 shows how kernel extension pre-processor modifies kernel extensions at binary code level. The relocatable ELF object file is general form of kernel extension. Two prerequisite information for modifying kernel extension are a symbol table represented by *.symtab* and a string table represented by *.strtab*. These information can be easily obtained from ELF object file itself without disassembler or particular tool such as *readelf*[12]. From reading ELF object file and parsing the ELF header and section header table, location of *.symtab* and *.strtab* are obtained. *.symtab* is a table of symbol entry encoded as binary. However, *.strtab* is a set of strings delimited by null character and encoded as ASCII. For this reason, kernel extension pre-processor knows what kernel routines are used by kernel extension.

Once kernel extension pre-processor analyzes function names to intercept in *.strtab*, it appends function name of wrapping stubs with a prefix: 'sentry_' and then modifies name field of each entry of *.symtab*. That is, each entry of *.symtab* has name filed which is an offset value from start of *.strtab*. In Figure 4, dashed line means original function name in kernel extensions. After attaching wrapping stub function names, each symbol name filed is assigned newly calculated offset. This process makes it possible that modified kernel extension does not call original kernel routines, but Sentry's wrapping stubs.

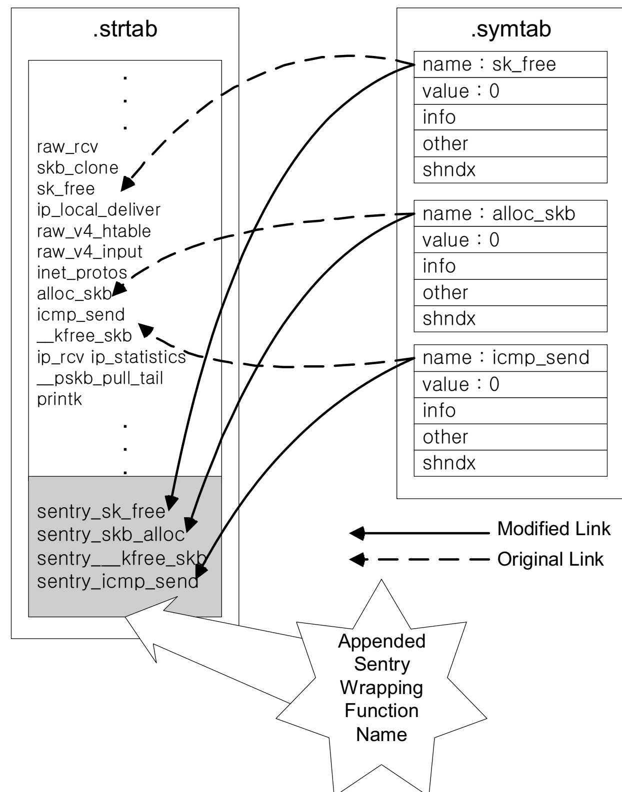


**Figure 4. Relocatable ELF File Modification**

Relocation process is necessary for kernel extension to use published kernel interfaces. In Linux, if you issue *insmod* to load kernel extension, kernel module loader copies kernel extension into kernel address space and relocates unresolved symbols. To relocate relocatable ELF object file, symbol table, relocation entry, and string table are necessary. Relocation entry holds information that describes how to modify their section contents and consists of *r_offset* and *r_info*. *r_offset* gives the relocation at which to apply the relocation action. *r_info* gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. Symbol table represents a program's symbolic definitions and references like functions and variables. For example, if kernel extension calls *alloc_skb()* functions fifteen times, there are fifteen relocation entry about *alloc_sbk()* and one symbol table entry represents *alloc_skb()*. If referenced symbol is not defined in kernel extension code, kernel module loader should search where the location of reference symbol is by comparing symbol name. Because using kernel extension pre-processor, symbol name are modified,


IEEE COMPUTER SOCIETY

kernel module loader should bind address of symbol to Sentry's wrapping stub instead of original functions. This is the reason that Sentry can insert function hooks and interpose kernel extension without source code or assembly code of kernel extension.

Modifying relocatable ELF object file is more efficient than other mechanisms presented in previous works. Because, in case of same symbol, address relocation process can be done at every places referenced by every *r_offset* field of relocation entry. For example, *tcp.o* in Linux kernel version 2.4.18 has 13 relocation entries corresponding to one symbol: *__kfree_skb()*. To hook all *__kfree_skb()* functions, HECK or SFI will insert hooking code at 13 times. Sentry, however, just appends new name string which is *sentry__kfree_skb* and modifies name field of symbol entry in *.symtab* section.

# 4. Policy

## 4.1. Pole of Policy

Policy participates in operations of two parts of Sentry: Sentry Run-time System and Extension pre-processor. Using policy, Sentry Run-time System decides whether allowing kernel extension's function call or not. If the amount of request for kernel resource of kernel extension exceeds limitation specified in policy, Sentry Run-time System does not accept requests any more.

Policy maintains a list of function names to intercept. When Extension Pre-processor tries to modify and find function names, it lookups function name list in policy. If function name is in policy, it replaces old name to new function name.

Policy specifies that what kind of kernel routines and how many kernel resources are allowed to kernel extension. Policy of Sentry can be divided into 1) behavioral policy and 2) quantitative policy. Behavioral policy describes that what kind of kernel routines are allowed and it defines calling authority of each kernel routines such as *alloc_skb()*, *printk()*, *register_blkdeb()*, etc.

Quantitative policy describes that what kind of resource and how much it will be allowed to kernel extension. There are several kinds of kernel resource such as socket buffer, kernel memory, timer, etc. Sentry provides quantitative policy for *struct sk_buff*, dynamic kernel memory and I/O port. Since Linux provides routines for these kernel resource such as *allock_skb()*, *kmalloc()*, and *readb()*, Sentry can intercept these routine call.

## 4.2. Design of Policy

There are two design issues of policy. First, Sentry Run-time System and Extension Pre-processor should access policy. However, they execute in different privileged level and address space. As described in section 3, Sentry Run-time System runs in kernel address space and Extension Pre-processor executes in user address space. To share policy between Sentry Run-time Sys-tem and Extension Pre-processor, it is exported by Linux /proc file system. Once Sentry Run-time system and wrapping stubs are loaded, new policy is created and then it is exported through /proc file system. Be-fore Extension Pre-processor modifies kernel extension binary code, it read /proc/Sentry_policy and replace target function name. Figure 5. shows example of /proc/Sentry_policy. It consists of behavioral policy and quantitative policy. Behavioral policy describes function call authorities. It has several entries for each function name. Each entry has two-tuple: <authority>, <function name>. In figure 5., kmalloc() function will be allowed, but request_region() and release_region() will be rejected.

```
$Behavioral Policy
permit kmalloc
permit kfree
permit alloc_skb
permit __kfree_skb
reject request_region
reject release_region
... ...

$Quantitative Policy
limit kernel_memory 2000K
limit IO-port 0x1800 0x181f
limit sk_buff 3000K
```

**Figure 5. Example of /proc/Sentry_policy**

Second, how does Sentry build up policy and apply it to execution of kernel extension? This means that policy is just hope of human who expects that Sentry will allow some function call and reject other call. However, Sentry does not understand each meaning of policy. For example, there is *sentry_kmalloc()* which is wrapping stub for *kmalloc()* and policy about *kmalloc()*; `permit kmalloc`. If *sentry_kmalloc()* is called by interception of function call of *kmalloc()*, it queries whether *kmalloc()* is allowed or not. Since Sentry does not know that *sentry_kmalloc()* is wrapping function for *kmalloc()*, relation between these original function and wrapping stub should be described. Moreover, though *kmalloc()* is allowed, if


IEEE COMPUTER SOCIETY

kernel extension already allocates kernel memory up to limitation, Sentry must reject this function call. To solve this problem, we define additional data structures.

```
#define ACCESS 0
#define ALLOC 1
#define FREE 2

#define PERMIT 1
#define REJECT 2

struct policy_entry {
    char* function_name;
    u16 authority;
    u16 action; /*may be ACCESS, ALLOC,
or FREE*/
}
```

**Figure 6. *struct policy_entry***

*struct policy_entry* describes behavioral policy. It consists of three member variables. Figure 6. depicts *struct policy_entry*. `function_name` represent name of function which is intercepted by wrapping stub. `authority` is permission of hooked function. It can be either `ALLOW` or `REJECT`. `action` means that this function will access, allocate, or free the kernel resource.

## 5. Conclusion

This paper proposes Sentry; a lightweight kernel subsystem that provides dependable execution environment for the kernel extensions. Sentry modifies Relocatable ELF object file to interpose kernel extension's function calls. This mechanism does not require any reverse-engineering tools such as *objdump* or any dissassembler. Therefore Sentry does not compile source code or assembly language. This feature is very attractive resource restricted system like embedded system. Because embedded system usually does not have any disassembler, compiler, and libraries, it can not compile modified kernel extension code. However, our mechanism does not compile kernel extension code. In despite of simple mechanism, it can capture the whole function call of kernel extension and monitor kernel extension's behavior.

Sentry has flexible structure of its run-time system. Since we separate wrapping stub and other run-time system, we can add or remove any functionality of Sentry at any time. To help implementation of wrapping stub, several library routines which interact wrapping stubs and run-time system, are provided. Each wrapping stubs can be implemented separately and loaded into kernel address space by kernel module loader.

## References


[1] J. Corbet, A. Rubini, and G Kroah-hartman, "Linux Device Dirvers", 3rd Edition, O'reilly, Feb 2005.

[2] R. Wahbe, S. Lucco, T. Anderson and S. L. Graham, "Efficient Software-Based Fault Isolation," 14th ACM Symposium on Operating Systems Principles, Ashville, NC, Dec. 1993.

[3] J. Liedtke, "On µ-kernel construction", In proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 237-250, December 1995.

[4] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian, "Mach: A new kernel foundation for UNIX development", In Proceedings of the 1986 Summer Usenix Conference, pages 93-113, June 1986.

[5] Executable and Linkable Format (ELF). Version 1.1, TIS Committee, Oct. 1993.

[6] Haizhi Xu, Steve C, and Wenliang D, "Detecting Exploit Code Execution in Loadable Kernel Modules", In Proc of the 20th Annual Computer Security Applications Conference, Dec 2004.

[7] Rob Short, Vice President, of Windows Core Technology, Microsoft Corp. private communication 2003.

[8] Micheal M. Swift, Brian N. Bershad, and Henry M. Levy, "Improving the Reliability of Commodity Operating Systems.", In Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 2003.

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

[10] 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.

[11] K. Elphinstone and S. Götz, "Initial evaluation of a user-level device driver framework", In 9th Asia-Pacific Computer Systems Architecture Conference, Beijing, China, Sept. 2004.

[12]GNU Binutils http://www.gnu.org/software/binutils/

[13] Micheal B. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface", In Proceedings of ACM Symposium on Operating System Principles, page 80-93, December 1993.





IEEE COMPUTER SOCIETY