On Supporting Per-Process based System Call Vectors in Linux Kernel

Mallesham Dasari, Erez Zadok
Stony Brook University, Stony Brook, USA

Abstract

Linux has a single global system call vector which is accessed by all the processes that invoke any system call. It prohibits a loadable module from changing the global system call vector. Linux makes it even harder to modify system calls, either globally or on a per process (or process group) basis unlike dynamic overriding of system calls in BSD. Maintaining the per-process based system call vectors is a valuable feature for many processes. For instance, we can restrict a process from invoking system call it should not (unless it was hijacked). In this paper, we present an efficient way (through loadable modules) to support this feature in Linux 4.6+ kernel. In this method, each process will have its own system call vector and can change it as and when required. Further, a process can acquire multiple system call vectors along with a default vector, if required. Our design involves very few lines of code (A maximum of 10 lines added) change in Linux kernel. The experimental evaluation shows that our system is highly robust and versatile with small overheads.

1 Introduction

User processes request a kernel service through the means of system calls to perform some privileged or time critical tasks. On a high level view, as soon as the user process issues a system call, the CPU will suspend the user process and switches from user mode to kernel mode and executes a piece of kernel code. Once the kernel routine is completed of its execution, the user process is resumed for continuing further and is returned output of system call. Traditionally, in Linux, there exist a Global System Call Vector/Table (GSCV) [7] which maintains all the system calls defined in Linux with unique System Call Number (SCN) for each system call. Linux prohibits the user/kernel processes to override the GSCV and hence a simple change in any system call code will lead to recompiling the kernel to reflect the changes. Further, Linux allows every user process to access the any system call without any background checking of process. This can lead to many issues such as security, protection and accessibility of system calls. For instance, a malicious user can easily access the crucial system calls very rapidly so that the legitimate processes will experience higher delays in having the kernel service or may lose complete control of system calls [3]. Or a network related process such *httpd* is nothing to do with *mkdir*, unless it was hijacked and hence it should be prohibited from the accessing it. Embedding this kind of process specific information in system call definition is not scalable as each process has to go though this phase. Hence, these difficulties can be overcome very easily if we support the system calls not on a global basis but on per-process based. A simple flow of a system call is shown in Fig. 1 using global and per-process methods. Fig. 1(b), shows multiple SVCs can be loaded/unloaded into kernel using Loadable Kernel Modules (LKMs). This method alleviates a process from calling prohibited system calls and provides security and protection for any system call on a per-process basis.

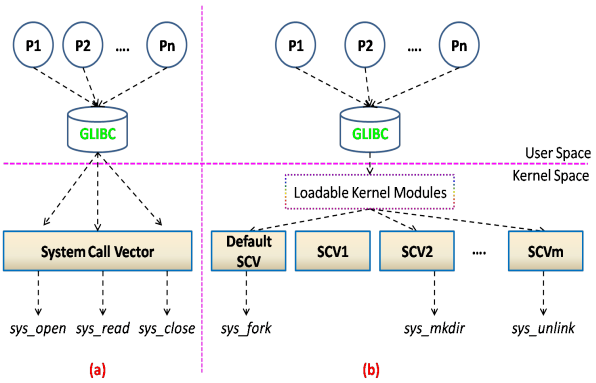

Figure 1: **Global and Per-Process Based SCV**

2 Background and Motivation

A user process invokes a system call using GlibC as shown in Fig. 1. GlibC packs the arguments of a system call into registers and invokes the kernel routine. A trap is issued to Operating System (OS) using specialized assembly instructions. Upon interrupt, a system call handler held by GSCV, is called using unique __NR_SYSCALL number, to perform the kernel service. This number is extracted from *rax* register. The arguments are extracted from the registers *rax, rbx, and rcx etc*. After the completion of system call, the return value is sent to user process by again using a register. Here, in this process, we can easily intercept the system call invocation and can change the system call handler using LKMs, as and when needed. Fig. 1 shows the newly created, modified, overridden and customized system calls in the form different vectors SVC1, SVC2, ... SVCm, where m is the number of SVCs in the system at current state.

Pradeep et.al [8] and Ram et.al [5] have done a similar work in the past. The authors proposed a method to intercept the system call at the entry level. This means, for every system call, the new change has to incorporated which imposes lot of overheads. In [1], Don came up with a per-process based system call monitoring, where we trace the highest number of system calls invoked by a process in a given time. Also, this method has many system trap calls to display different statistics of the system. In [2], a systematic overview of the evolution of system call monitoring is studied. Here, the authors came up with different ways tapping system calls and prohibiting them from access. In [4, 6], the authors proposed a method to detect the anomalous system calls. These methods tweak each system call or inserting a piece of code before calling the system call in the GlibC or at the entry level. However, this introduces lot of overhead where this operation is performed even for all the the system calls which are not required to be monitored. And, to our knowledge, there is no prior work on maintaining a per-process based SCVs. In this work, we introduce an efficient way to create, modify, override and wrap the system calls on a per-process basis. And, we make this method possible without the need to change to kernel as and when a process need to change the SVC. We consider as an important design paradigm for upcoming Linux kernels with the following applications taken as motivation:

- A user process can change the semantics of its own specific system calls without affecting the behavior of other processes and without changing kernel
- Profiling and Logging of process specific system calls is made easier
- A specific set system calls can be allowed to a specific process. In other words, some processes can be prohibited from accessing the protected system calls without changing the kernel
- Flexible way allowing kernel data structures to be modified by a user process

3 Design

We considered the following five design goals:

Flexibility: This is an important feature of our design. Unlike other works [8, 5] and BSD, we allow the user processes to change the SCVs as and when needed. We designed it to support multiple operations: add, delete, count and list the SCVs on per-process level.

Scalability: Scalability is yet another crucial feature in system call flow. We can add any number of SCVs through LKMs, without affecting the performance of other processes. It also does not involve any change of kernel, whenever we add a new vector. This is an interesting feature of our design.

Availability: Availability is addressed in terms of default SCV. This is because, suppose there are no SCVs registered in the system but a user process wants to access some basic system call. There is a default SCV which consists of all the system calls that are allowed by all the user processes.

Performance: It important that changing the design of a system call flow should not incur too much performance overhead. Here, we limited the intercepting of system call to a process specific and system call specific. This alleviates the huge overheads incurred when we intercept at the entry point level or GlibC level.

Security and Privacy: It is essential that the newly created, modified and overridden system calls must ensure security and privacy towards user level processes. Hence, we enforce process specific information in the LKMs and check if the processes are allowed to execute the system calls or not.

In Section 3.1, we discuss about where to intercept the system call, and in Section 3.2 we describe the component architecture our method.

3.1 Intercepting the Syscall

The main idea of our work is to provide a per-process based system calls. This is in the form of new, modified, overridden and wrapped system calls. The crucial component of this design is where to intercept the original system calls. There are multiple ways of intercepting the system calls and diverting them from the original calls. They are:

- **GlibC Level:** A system call can be intercepted at the GlibC level by tapping the arguments and system call number. However, this is a hectic task which involves lot library changes every time we add a new SCV.
- **Entry Level:** A system call can be intercepted at the assembly level at the actual entry point of a system call in the entry.S. Before calling the *sys_call_table*, we can save the registers on the stack and invoke our middle-ware. Upon completion of middle-ware task, we can restore the registers by popping from the stack and decide to invoke the actual call or not. However, it can be easily observed that this method induces severe performance degradation due to unnecessary middle-ware calls before non-process specific system calls.
- **System Call Level:** We can modify the behavior of system calls by inserting process specific code in each system call. For example, if we do want an *httpd* process to access *mkdir* call, we can just check if it is this process inside the *mkdir* call, and can restrict it. However, one can easily see the tedious change of each system call which consumes huge lines of code
- **Module Level:** A specific system call can be intercepted using an LKM by acquiring the *sys_call_table* inside a module. As only a required system call is being intercepted, this method is highly flexible and scalable. The overhead is very less because it does not affect other processes and system calls. Further, this method involve very few lines to be added Linux Kernel

3.2 Component Architecture

The flow of system calls is changed because of multiple SCVs as shown in Fig. 2. It consists of four critical components: SCV driver, System Call Intercepting (SCI) driver, Helper driver and userland module. *SCV driver* creates abstractions for registering and unregistering SCVs. *SCI driver* intercepts the system call and diverts it to specified SCV of a user process. The *Helper driver* provides the APIs for changing the kernel code at

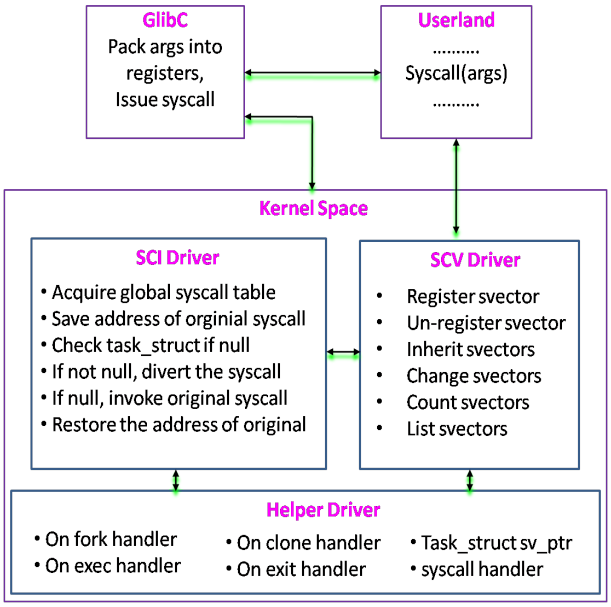

Figure 2: **The Complete Picture of System Call Flow**

the boot time. Finally, *Userland* issues an SCV based system calls and tests if they are calling the default system call or process specific system calls. The combination of these modules creates the per-process based SCVs. We allow a process to change its SCV any number of times. A process can have more than one SCV and including default SCV. The simplicity and scalability of our design is highlighted in writing the APIs for these drivers. We created the abstractions in such a way that it can be easily extended and ported on to any platform. It is also highly architecture independent that require no change of lines of code.

3.3 SCV Driver

SCV driver follows the typical Linux syntax and semantics in handling the operations. It has *register_svector* and *unregister_svector* APIs whenever an LKM wants to insert and delete the SCVs respectively. This driver interacts multiple LKMs to have custom SCVs from different users. Whenever an LKM invokes the *register_svector*, it inserts the SCV into a list along with its name and id. Similarly, in *unregister_svector*, it deletes the entry of SCV by looking for its name and id. It also interacts with user processes as shown in Fig. 2. It supports operations such as:

1) List the number of SCVs present in the system
2) List the number of SCVs for a given process
3) Count the number of SCVs
4) Load a new SCV for user process

5) Change the SCV of a user process
6) System call inheritance from parent to child process when forked, cloned and execed and exited etc.

Once an LKM registers with SCV driver, and whenever a user process sets its SCV corresponding to that module, its reference count is incremented. Unloading this module is prohibited until the reference count is zero i.e, all the processes that are registered with this vector must be exited. Further, it maintains separate statistics sub-module which records total number of processes accessing the SCVs, total number of SCVs present at a given time and traffic incoming and outgoing from the user land to kernel space.

3.4 SCI Driver

Once the SCV driver is setup with few vectors or atleast with default SCV, the SCI driver is initialized. Its functionality is divided into two phases. It does not have any work until a user process invokes system call. We made this driver passive to reduce the huge amount overheads induced with other approaches such as entry level and system call level intercepting. We modified default SCV of Linux from its constant type to our customized type so that we can modify the function pointer of system call handler of this vector. This is three step procedure: First, we save the address of the original sys handler into a temporary pointer, then we give this handler to another module which decides which system call to invoke (phase2) and finally, the original system call address is restored on completion of system call before returning to user process.

In the second phase, through the use of Helper driver, it acquires the *task_struct* of the user process which invoked this system call. It checks if the SCV field of this task is empty or not. If this field is empty that means the process do not have any specialized SCV and hence invokes default system call. If the field is not empty, then it goes to corresponding SCV and invokes respective system call.

3.5 Helper Driver

The helper driver communicates with both SCV and SCI drivers. It provides the APIs to change the Linux source. For instance, to change the SCV field of *task_struct* of user process. Whenever a process sets its SCV, the SCV driver communicates with this helper driver to change the SCV field this particular process. This will used in invoking the actual or overridden system call in the SCI driver. Similarly, it plays a key role in SCV inheritance explained in Section 3.4. It provides abstractions to modify the fork, exec, clone and exit cousins of Linux multithreading mechnanism. We created new version of clone to support for it's child to have a designated SCV. Further, it has modules such as *netlink_sockets* and system tap functionality to monitor the volume of system calls invoked on per-process basis.

3.6 Userland

A specific set of user processes are created to test the functionality of the per-process based sytem calls. First, we created a process just to load and unload its own SCV multiple times. Second, we created another user process which tracks the number of SCVs and lists them on a per-process level and in total. Third, we created several processes to check the flow of per-process based system calls which tests the newly created, overridden, wrapped, customized and modified system calls. Finally, the interesting processes which invovle lot fork, exec, clone functionality, are created to test the child processes have inherited its parents' SCVs.

Although the design specifically focuses on per-process specific system calls, these modules together forms a numerous features:

- **System Call Overriding:** Using SCI driver, the system calls can easily be overridden. This can be done even at the runtime without affecting other user processes.

- **System Call Wrapping:** Suppose if the users does not want override the system call but enforce a specialized check (e.g. security check) before invoking the system call. Here, the SCI driver gives us provision to wrap the system call in a new function and then check if the process has privileges to execute the system call or not.

- **System Call Inheritance:** This is interesting result of our design. We inherit the parent process's SCV to its child process whenever a fork is called. When a clone system call invoked, we check the clone flags to check if child process should inherit or not. If not, we use default SCV for child process. Further, a new version of clone is designed in such a way that it accepts the SCV name and id from user process and Helper driver sets the designated SCV to child process. For all the newly created child processes, a simple exit handler API is provided by Helper driver to decrease reference count of child processes so that the registered LKMs would eventually be unloaded.

4 Implementation

We developed a prototype of our design as multiple LKMs using Linux 4.6+ kernel. The latest prototype has exactly 988 lines of code along with a 14 lines of code added in the Linux source. We now descibe the few interesting aspects of our prototype implementation.

4.1 Registering the Framework

The SCV registration is divided into the following: 1) Adding System calls to SCV and 2) Registering the SCV. Each LKM defines set of system calls which newly created, modified orginal system calls or overridden system calls. We created an abstraction to add there custom system calls to a given SCV name and id. Hence, the same abstraction is used for all the LKMs that define multiple SCVs. Multiple new system calls are defined such file merge, file sort system calls etc. Once, these system calls are packed into the vector, the SCV is registered with SCV driver using *register_svector* API. Each SCV contains the information such as: Name of the vector, reference count, address to the vector and a boolean array which indicates which system calls overridden, wrapped or modified.

Once the SCV driver is setup, the user processes are communicated with SCV driver in several of the following ways:

1. Ioctls
2. Prctl
3. System Call
4. Netlink Sockets
5. Sysctl/sysfs
6. Additional exec system call

We used ioctls and a system call which are easy to use, flexible for extention and does not incur lot user-kernel communication overhead. Further, we developed *netlink_sockets* based interaction inorder for the kernel to initiate the action instead user process. This is very useful if kernel want inform any information to a specific user process.

4.2 Reference Counts

As described in Section 3.2, we maintain the reference counts (rc) for each LKM. This tells the number of user processes which are using an LKM at present. The value of rc increased by one when a user process loads corresponding SCV. Similarly, the rc is decreased on process exit. Suppose if an LKM try to exit in the middle of a user process execution, it can prohibited by using *MODULE_PUT* and *MODULE_GET* methods with the rc.

4.3 Locking Semantics

Locking is the critical component of our design. On a multiprocessing system, our design leads to inconsistent results without enforcing locking semantics. We used *atomic_t* for reference count as it is a single shared variable. Hence, locking on rc is performed internally. For accessing the list of SCVs, we read write semaphore. This is very helpful becuase, we allow many processes to read the list of SCVs so that counting, listing operations are not blocked. Hence, multiple processes can access this list by grabbing the read lock. We take write lock while modifying the list of SCVs so that all the processes are synchronized.

4.4 Module Validation

We restrict the SCV developer from overriding fork( ) and exit( ) system calls as it can break the existing functionality and also disturb the reference count values of the vectors. Hence any loadable module which tries to support a new system call vector that is overriding fork or exit will get an error during loading of module.

5 Conclusions

In this work, we descibed a framework to override the default SCV in Linux 4.6+ kernel. This feature gives the user process provision to dynamically select a new SCV thus making it feasible to have multiple functionalities from the ones present in the default SCV. We provided this framework in the form of a system call which the user process can invoke to set its own SCV.

References

[1] DOMINGO, D. Per-process based system call monitoring. `https://sourceware.org/systemtap/SystemTap_Beginners_Guide/index.html`, 2010. Accessed: 2016-12-04.

[2] FORREST, S., HOFMEYR, S., AND SOMAYAJI, A. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 418–430.

[3] MUTZ, D., VALEUR, F., VIGNA, G., AND KRUEGEL, C. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC) 9*, 1 (2006), 61–93.

[4] PROVOS, N. Improving host security with system call policies. In *Usenix Security* (2003), vol. 3, p. 19.

[5] RAM, MARUTHI, H. Per-process based system call inheritance. `https://github.com/rangara/syscall-inherit/blob/master/hw3/design.txt`, 2012. Accessed: 2016-12-02.

[6] ROSENBERG, I., AND GUDES, E. Evading system-calls based intrusion detection systems. In *International Conference on Network and System Security* (2016), Springer, pp. 200–216.

[7] RUBINI, A. Tour of the linux kernel source. *Linux Documentation Project* (1994).

[8] SHUKLA, A. Per-process based system call inheritance. https://github.com/abhishekShukla/System-Call-Inherit, 2012. Accessed: 2016-12-02.