

Systemcall Inheritance

Ram, Maruthi and Hema
Stony Brook University

Abstract

In the Operating System world, system calls are a way for a user process to request services from the kernel. When a process makes a system call, a context switch happens and the control shifts to the kernel. In Linux, the system calls are implemented through a predefined system call handler. There exists a default base system call vector which holds the pointers to the different system calls. Unlike Linux, BSD supports dynamic overloading of system call vectors. In this paper, we present a way to support this feature in Linux 3.2. Using our implementation, a process can set its own system call vector which has its own system call implementations. Our design incorporates new system call vectors through loadable kernel modules and we provide a system call interface for the user process to dynamically change its system call vector. The complete details of our design and implementation are covered in the next sections.

1 Introduction

In the world of operating systems, system calls are a way for a user process to request a service from the kernel. Most of the major user level operations such as opening a file, reading from/writing to a file, forking a process require a user process to request these functionalities from the kernel. System calls are a means to do this. A user process makes a system call when it needs a service from the kernel. At this point, a context switch happens and the controls shifts to kernel. There are many variations in the way the system calls are supported in various operating systems. For instance, the BSD operating system provides to its user processes, flexibility for dynamic overriding of the system call vectors. In contrast to BSD, the Linux operating system only supports a single predefined system call vector `sys_call_table` which is statically loaded during the kernel boot-up. Every system call in Linux corresponds to a handler and the set of all these handlers is stored in the system call vector indexed by the system call number. When the user process makes a system call by providing the system call number, the handler at the index equal to this system call number in the `sys_call_table` vector is invoked. Currently there is no way in Linux, to override the default system call vector.

In this paper we set out to explore the ways to support system call inheritance. Section [2] gives a brief background and motivation for our work. We present our de-

sign details and the various possibilities which we have considered during our design in section [3]. In section [4] we present our implementation details. Sections [5] and [6] discuss the various use cases which can be supported by this feature and the related work in this area. Finally, we conclude in section [7] followed by our future work details in section [8].

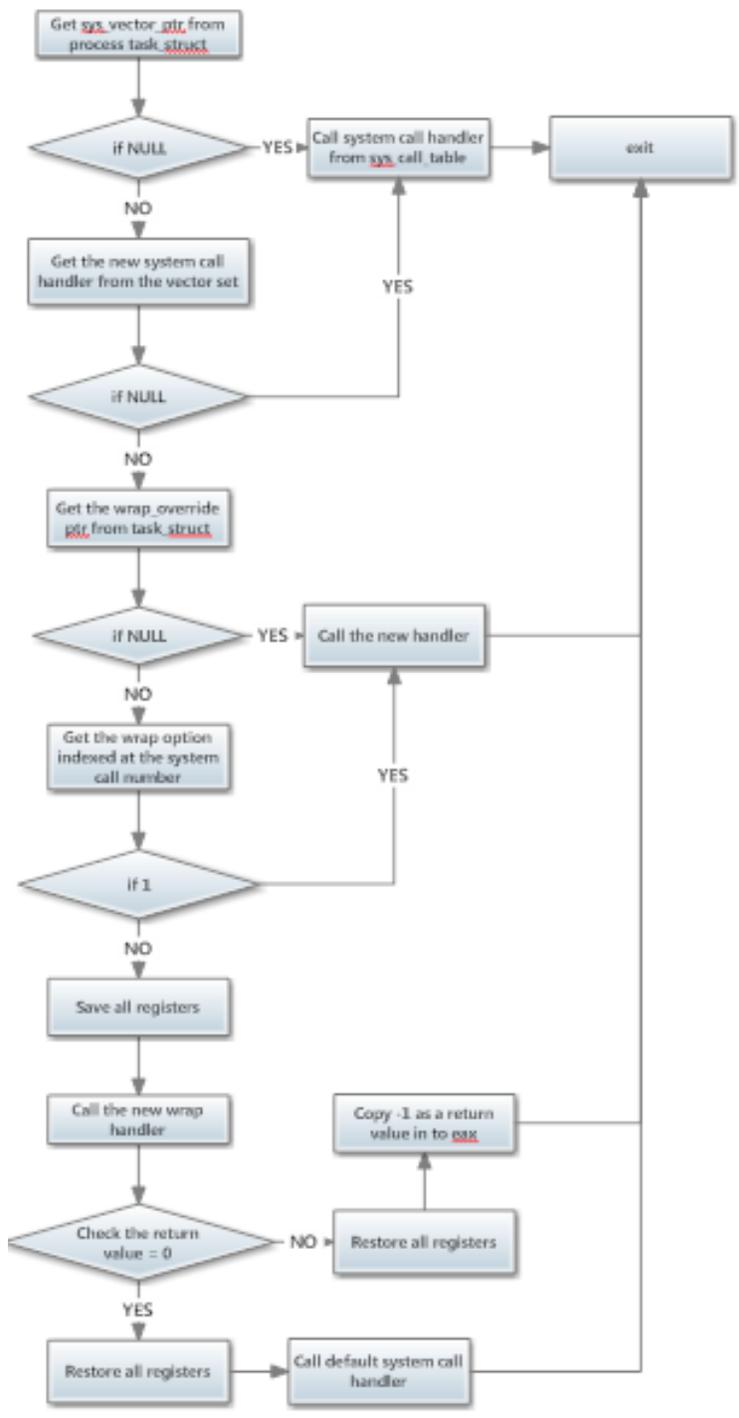
2 Background

In Linux, the system calls are implemented through an interrupt mechanism. There is a unique number associated with every system call. Every system call will have its own predefined handler. The default system call vector `sys_call_table` holds these handlers which are indexed by the system call number. When a user program issues a system call, it in turn calls a library routine. The library routine issues a trap to the Linux operating system by executing `INT 0x80` assembly instruction. The system call number and the arguments of the system call are passed to the kernel by storing them in `eax` and other registers (`ebx`, `ecx`, etc.) respectively. The kernel then executes the system call by invoking the handler obtained from the `sys_call_table` and returns the result to the user program using a register.

3 Design

The main idea of our design is to support new system call vectors as loadable kernel modules (LKM). Any new vector which needs to be added should be implemented as a LKM and inserted into the kernel. Since the process `task_struct` is unique for each process, we decided to store the vector information in the `task_struct` of the process which can be accessed from anywhere in the kernel using `current` pointer. New fields are added to the process `task_struct` to incorporate this feature.

We have defined a framework `sys_set_syscall_vector` which is central for our design. This framework holds all information about a newly added vector, its reference counts and it does the updating of the process `task_struct` when a process requests to set a custom system call vector. From here on, we refer to this framework as `sysvec-framework`. Every vector newly inserted has to register with this framework. This is required as the process needs to know the list of currently supported custom vectors in the system. The framework also handles the reference counts of the vectors. A reference count (`rc`) needs



to be maintained for a vector in order to avoid removal of the vector when a process is using it. A vectors reference count increases by one in two scenarios

1. when a process sets this vector in its process task_struct
2. when a child process inherits this vector from its parent through fork()

A vectors reference count decreases by one in one of the following two scenarios:

1. When a process exits, the rc of the vector is decremented. Updating the reference count requires locking on the rc variable in order to have synchronization among multiple processes.
2. When child wants to set its own vector, the refcount of the vector it inherited is decremented.

We limit the number of times a process can set its vector to one. Any further attempts by the process to set its vector will return an error. This is to avoid any malfunctioning of the system calls. When a process is fork-ed(), even though the child process inherits the vector set by its parent, it can still set the vector of its choice through the framework once. This information about whether the process has already set the vector once is stored in the process task_struct structure.

As an added feature we are also supporting two types of system calls, wrap-systemcall and override-systemcall. A wrap-systemcall acts like a wrapper to the default system call. It can be used to validate the inputs before invoking the actual system call. An override-systemcall completely overrides the default system call. As a custom system call vector developer, one can decide whether a system call in the new vector should behave like a wrap-systemcall or an override systemcall. The vector will pass this information to the framework during the registration which will be set in the process task_struct when the process sets this vector. It is up to the vector developer to decide whether to support the wrap/override option or not. Not setting this option implies that all the system calls supported by this vector are by default overriding system calls.

Following is the information per vector which is maintained in our framework when a vector registers

1. Name of the system call vector
2. The reference count of the vector
3. Address to the system call vector
4. Address to the boolean array

As we have seen in the previous section, when a system call is invoked the system call number is stored in the eax register and the handler in the sys_call_table at the index same as the eax value is invoked. Now to incorporate our design we modify the flow as follows

- before invoking the system call handler of the sys_call_table, we will check to see if the vector pointer

in the process task_struct is set and if not the default system call is invoked

- if the vector pointer is set, then check if the new handler for the invoked system call in this new vector is set. If not, the default system call will be invoked
- if the new handler is set and
- the wrap-array in the task_struct of the process is not set , invoke the new handler
- the wrap-array is set and the type of system call in the wrap-array is override-systemcall (0), invoke the new handler
- the wrap-array is set and the type of system call in the wrap-array is wrap-systemcall (1), then invoke the new handler first and on its return, based on the return value either call the default system call (if the return value of the wrapper system call was a success - 0) or return -1 to the process that invoked the system call.

4 Implementation

The implementation details of our design are explained in this section. We worked on Linux 3.2.2 version.

4.1 Adding a new system call vector

The first step of our approach is to add a custom system call vector which is implemented as a loadable module. This module consists of the system call vector whose size is the same as the default sys_call_table and which holds the handlers for the newly overridden system calls. The implementation of these handlers is also done inside the module. The handler for the system calls which are not being supported by this vector are set to NULL. The vector that wants to support the wrap/override option can do so by defining a new boolean array of size same as the sys_call_table. A value of 0 in this array indicates an overriding system call and 1 represents a wrapper system call. Following is the information per module which needs to be stored when the module is inserted:

1. Name of the system call vector (as listed to the user process)
2. The reference count of the vector (indicates the number of processes currently using this vector)
3. Address to the system call vector (pointer to the new system call vector table)
4. Address to the boolean array (pointer to the wrap/override array which can be NULL)

We have defined a new structure sys_vector_data which represents the vector data mentioned above.

```
struct syscall_vector_data{
    atomic_t refcount;
    char name[VECTOR_NAME_LEN];
    void *vecptr;
    int *b_wrap_override;
    struct list_head mylist;
};
```

4.2 Registering with the framework

In section 3, we have discussed how to create a new system call vector as a loadable module and that the information about all such vectors must be stored. This module provides an API `add_syscall_vector` which should be called by a vector to register itself with the framework. Our framework maintains a list `sysVectors` to hold the pointers to `sys_vector_data` corresponding to all the existing system call vectors. Whenever a new system call vector module is inserted, a new pointer to `sys_vector_data` is created and filled with the vector data and added to this list during registration.

4.3 Listing the system call vectors

The central interface for our approach is provided as a new system call `sys_set_syscall_vector` (`int option, void* data`) which has been implemented as a loadable module. From now on, we address this system call as `SET_SYSVEC`. The data pointer parameter acts as both in and out parameters depending on the option value. `OPT_VECTOR_COUNT` and `OPT_VECTOR_LIST` are options that must be used together. `OPT_VECTOR_COUNT` returns the number of syscalls currently loaded. `OPT_VECTOR_LIST` returns a (`char **`) of vector names. This (`char **`) buffer is allocated by user program based on the number of syscall vectors loaded. We copy the vector names into the user buffer by `copy_to_user()`.

4.4 Setting a system call vector

Any process which needs to override the `sys_call_table` has to first set the vector in its `task_struct`. There are many ways this can be supported namely,

1. Using `ioctl` - For each loadable module which is adding new system call vector(s), we can define an `ioctl` such that the handler for this `ioctl` is implemented in the corresponding module. Any process which wants to set a new system call vector can call the corresponding `ioctl` to do so. But the drawback with this approach is that we will end up defining a new `ioctl` for each new loadable module which is an overhead.

2. Defining a new system call - We can define a new system call which when invoked by the process with the vector name as input, will set the corresponding vector for the process.

3. Through `exec()` system call - A new wrapper call can be added for the `exec()` system call which in addition to the actual `exec()` system call arguments also takes one more argument which is the name of the system call vector.

For our design we have considered adding a new system call to set the custom system call vector. The `SET_SYSVEC` discussed in the previous section also handles the functionality of setting a new system call

vector. The process invokes this system call with the option value as `OPT_VECTOR_LOAD` and data pointer set to the name of the vector. Depending on the vector name, its corresponding vector is obtained from the `sysVectors` list and set in the process task structure.

4.5 Modifying the Process task structure

We have updated the `task_struct` structure to hold these additional fields: 1. `is_vector_set` boolean which indicates whether the process has already set its vector by invoking the `SET_SYSVEC` 2. `sys_vector_ptr` - address of the system call vector which the process wants to set 3. `wrap_override` - the address of the boolean array indicating the wrap or override option

We limit the number of times a process can set its custom vector by calling the `SET_SYSVEC` system call to one. To incorporate this, we have added the boolean variable `is_vector_set` in the task structure which if set to 0 indicates that the process has not set the custom system call vector and 1 indicates that the process has set its vector. When `is_vector_set` is 1, the process is blocked from further setting its system call vector. For a fork-ed() process this value is by default set to 0.

4.6 Invoking the system call

When the process invokes a system call, the flow will reach the `entry_32.S` file where certain modifications have been done to incorporate our feature. As mentioned before, the system call number is stored in the `eax` register and the parameters of the system call are stored in the remaining registers. We have added support in `asm-offsets.c` to generate the new `MACROS TL_task, TS_sys_vector_ptr` and `TS_wrap_override`, which represent the offsets of the `task_struct` in the `thread_info` structure, `sys_vector_ptr` pointer in the `task_struct` and `wrap_override` pointer in the `task_struct` respectively. Using these macros, we get the `task_struct` of the current running process and its corresponding `sys_vector_ptr` and the `wrap_override` pointers. And we do the required checking as explained in the design section.

4.7 Reference Counts

We have reference counts maintained for the vector which indicates the number of processes which are currently using it. The `rc` of a vector is incremented when a process sets it or when a child is forked from a parent which has set this vector. The `rc` is decremented when the process using this vector exists. Removal of a vector returns an error if the `rc` of a vector is greater than 0.

4.8 Locking

To achieve synchronization among processes we need to incorporate locking in to our framework. The reference count variable can be modified by multiple processes.

Hence this variable has been defined as of type `atomic_t` which handles the locking of reference counts internally. The list of vectors maintained in the `SET_SYSVEC` framework can be accessed by multiple processes at the same time. Hence there needs to be some kind of locking mechanism on this list. We have defined a read/write semaphore for this list. Any process which needs to access this list needs to grab this lock. Multiple processes can read from this list concurrently by grabbing the read lock but to update this list a process has to grab the write lock.

4.9 Module validation

We restrict the system call vector 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 `insmod`.

5 Usage

Below are the steps to be followed to test our implementation. The paths mentioned here are relative to the home folder.

1. Run `make` in `hw3` folder which will build all the LKMs.
2. Firstly insert the `sys_vector` module (`insmod`) which represents the system call added to handle the vector modules.
3. Insert the desired system call vector. Ex: `vector_logging.ko`
4. Run `make` in `hw3/usercode` folder which will build all the user files.
5. The user files expect the name of the system call as a command line parameter. For instance, to run the user file `logging.c` which needs to use the vector `vector_logging`, run the following command: `$./logging vector_logging`
6. If no vector name is provided, the default `sys_call_table` will be used.

6 Use Cases

The system call inheritance approach implemented in this paper can be used in two broad scenarios

6.1 Completely overriding the default system calls

As discussed in the previous sections, the vector developer can set an option for a system call to completely override the existing default system calls. This feature can be widely used in the following applications:

1. Restricting access to some system calls - There are certain cases, when we need some application not to

modify any of the files. One such example can be an application which spawns multiple children and execs new programs. If we have to block these spawned processes from creating/ modifying/deleting files, we can define a new system call vector which overrides the system calls `creat/write/unlink` which creates/modifies/deletes files respectively. In these overridden system calls we can just return the error messages. This way we restrict the application from doing something undesirable.

2. Customizing the default system call behavior - Some users do not want the default behavior of system calls. They want to implement their own functionality. This can be achieved by defining a new system call vector with customized system calls and setting this vector.

6.2 Wrapping the default system calls

Some applications might want to validate the privilege levels/parameters before invoking any system call. This can be done by defining a wrapper system call for the actual system call which needs to be called. Some such applications are listed below.

1. Logging of system calls - In some cases, the user might want to know what all system calls a particular application is using. In this case, they can define a new system call vector, with the system call handler set to their logging functions which will log the messages that a particular system call has been invoked and return success. The wrapping facility will take care of invoking the default system call after the user provided logging function has been executed. So if a user wants to know what all system calls a particular application is invoking, he can write a new program which will set this new system vector and exec the target application for which he wants to find out what all system calls it is using.

2. Access denial to system calls based on user privileges - There might be scenarios where access rights to certain operations like creation/deletion/updation of files are to be provided to root user alone. We can achieve this functionality with our approach. We can define a new vector by providing wrapping system calls for `creat/write/unlink` system calls which can verify the user who has invoked these system calls and return an error if the user does not have the required privilege levels. If the user does meet have the privilege levels, the underlying default system call can be invoked.

3. Denial of some system calls based on parameter validation - There are cases when `create/delete/ update` operations are to be restricted on some files/directories, for instance on the files in `/root` directory. The user can define a new vector providing wrapping system calls for the `creat/write/unlink` system calls which create/modify/delete files respectively. These wrapped system calls can check the parameters to see if the file passed is under the `/root` directory and if not, it can re-

turn error. In this way we can block modification access to certain files.

7 Related Work

A system call is the way in which the kernel services a program's requests. Each flavor of Linux would typically want to have its own wrapping/overriding mechanism to support custom operations. However Linux does not allow modules to override system calls, mainly keeping security in view. Most of the overriding in Linux is done using System call interception, but replacing the vector itself is not encouraged. BSD allows overriding by inheriting the system call dispatch vector per process. A policy can make a new process use an entirely new set of system call vectors. The system call vector contains a list of `sysent[]` entries whose addresses reside in the kernel module. Each `sysent` entry corresponds to a custom system call vector and contains the number of arguments, implementing function, audit events and general flags associated with the system call. When a system call is invoked, the trap code dereferences the system call function pointer off the process task structure.

8 Conclusions

In this paper, we provide a framework to override the default system call vector `sys_call_table` in Linux kernel 3.2.2. This feature provides the user process to dynamically choose a new system call vector thus making it feasible to have functionalities different from the ones present in the default system calls. We provided this framework in the form of a system call which the user process can invoke to set its own vector. Internally, our approach also supports both wrapped and overridden system calls which can be controlled by the vector developer. As discussed in the previous sections, various security applications can use this feature to impose access control rules on the users.

9 Future Work

There are certain aspects like the permission checks for setting the system call vector and customized error values for wrapped system calls and which we have not included in our current implementation. These features can be incorporated in our model in the future to improve the efficiency of our approach. The details of these features are explained below.

9.1 Access Control for System Call Vectors

Our current implementation does not perform any permission check on the user who is setting the system call vectors. Any user program can set any system call vector. We can implement access control for the system call vectors, to restrict certain user programs from setting certain vectors. The existing implementation can

be modified to maintain a group id/user id list for each system call vector, indicating which groups/users can access this system call vector. This can be provided while inserting the system call vector module. So when a user process tries to set a particular vector, its user-id can be checked in the list and if the id does not exist in the list, we can return an access denied error message.

9.2 Customized error for wrapped system calls

In the current implementation, even though the wrapped system calls return an appropriate error number, we are returning a standard error "-1" to the user program. We can support this feature by making changes to the assembly code to store the state of the program before calling the wrapped system call. Once done with the wrapped system call, if it returns a value other than 0, we change the stored stack state to contain the returned value, by popping `eax` from the stack state and pushing the return value in place of `eax`. Once the return value is pushed on to the stack, we can restore the stack state and go to the end section of the system call. For this we need to write a modified `SAVE_ALL` and `RESTORE_REGS` macros. The current `SAVE_ALL` macro pushes `eax` register first which makes it difficult to modify `eax` to contain the error value from wrapped system. To overcome this problem, new custom macros `MY_SAVE_ALL` and `MY_RESTORE_ALL` need to be defined which will push the `eax` value in the end and pop it in the beginning respectively. Thus the `MY_SAVE_ALL` will be pushing the error number present in `eax` on to the top of the stack and when we get an error value from wrapped system call, we pop the top value from stack and push the error number on to the stack. When we call our `MY_RESTORE_REGS`, it first pops the `eax` value, which contains the error message followed by the other register values. For this to work, we need to change the prototype of our new custom system calls to contain the system call number as the first argument because the top most value on the stack is the `eax`, which contains nothing but the system call number. We have implemented this in our current project, but it is not fully tested. Hence, the newly modified entry `_32.S` is not checked-in. Our future work would be to test this implementation.

10 Bibliography

1. Understanding the Linux Kernel. Authors: Daniel P. Bovet, Marco Cesati
2. An Advanced 4.3BSD Interprocess Communication Tutorial. Authors: Samuel J. Lefer, Robert S. Fabry, William N. Joy, Phil Lapsley
3. Design and Implementation of the 4.4 BSD Operating System. Authors: Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman.

4. Intrusion Detection using Sequences of System Calls. Authors: Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji, Dept. of Computer Science, University of New Mexico

5. Fine-Grained User-Space Security Through Virtualization. Author: Mathias Payer, Thomas R. Gross, ETH Zurich, Switzerland