Per-Process Custom System call vector support

Homework 3

CSE-506 (Fall 2016)

November 09, 2016

By Group 12:

Bharat Darapu (110949097)

Manish Kumar Valakonda (110945161)

Venkata Jyothsna Donapati (110950668)

Tarun Lohani   (110921666)

I) Abstract

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services, creation and execution of new processes, and communication with integral kernel services such as process scheduling. Generally, systems provide a library or API that sits between normal programs and the operating system and on Linux, that API is usually part of an implementation of the C library (libc), such as glibc, that provides wrapper functions for the system calls, often named the same as the system calls they invoke. Implementing system calls requires a control transfer from user space to kernel space, which is typically implemented using a software interrupt. Interrupts transfer control to the operating system kernel so software simply needs to set up some register with the unique system call number. For each of these system call, there is a unique system call handler.

This unique system call number is stored in the *"eax"* register of the processor before giving control to the kernel. The x86 instruction set contains the instructions *SYSCALL/SYSRET* and *SYSENTER/SYSEXIT* and each system call has to follow these. They represents the entry and exit of the kernel control of the system call. These are "fast" control transfer instructions that are designed to quickly transfer control to the kernel for a system call without the overhead of an interrupt. When a user process tries to execute a system call to get privileged access, a soft interrupt is generated at *0X80*. This interrupt indicates a system call and a context switch is triggered disabling all the other interrupts. Interrupt *0X80* looks up for the sys call table and tries to find out the system call based on the number stored in the *"eax"* register. Once syscall number is found in the table, its corresponding handler function is called and system call is executed. When the system call function returns, the control returns to the user space after executing SYSEXIT and all the interrupts are enabled again.

To modify a system call, we need to make changes in the system call table as well as handlers. Hence it is a bit difficult to change a system call as it requires kernel code changes which might introduce some serious issues. Our aim is to provide the users a way so that they can override the system call vector for each process. The subsequent child processes will also use that overridden vector by default. But in case the child wants to use its own system call vector, it can do so by calling a new version of clone system call which has been added. We created a module to register and deregister the vector names, which contain the new overridden system call numbers as well as their function implementation. The user can use the overridden system call vector through ioctl command. This will populate the private data field

of task_struct of the running process with the system call vectors containing the list of system calls and their respective function implementation.

II) Introduction

We cannot access the system call table in Linux through a user process. Also adding a new system call is difficult. We need to add the new system call entry in syscall_64.tbl and define the new syscall function handler in open.c. There are frequent use cases where the user want to add their function as a new system call. To make things more customizable, we wanted to define a vector of system calls per process. This vector of system call will have mix of original, overrided and wrapped system calls. This can be achieved by intercepting the slow path in x86 where the system calls are called.

We want to implement a per process system call where an application process can have its own vector of system call. The process should also be able to define their own implementation of those system calls. For example, an application while calling a system call might want to log the related information in a file. Hence we can wrap the original system call in a new system call which logs the related information and then calls the original system call. Similarly user might want some new functionality from the system call and want to override the original functionality. To achieve this, we create a vector with overrided and wrapped system calls which the process would like to use instead of original system calls. We set or remove the custom vector to a process using ioctl. If process calls a system call which is not present in its assigned vector then it will call the default system call.

We have modified the task_struct and added a pointer variable to the custom vector of system calls to be used by the process and the corresponding vector id. Hence this will maintain a per-process system call vectors information which can be used while calling the system calls. The process will have to choose the system call vector before calling any system call, through ioctl of the "ioctl device" character device loaded also via module.

A user process choose any one of the registered system call vector from the list of vector tables defined. This will populate the vector table address in the task_struct. Now whenever this process calls a system call, the process will be using the overridden or wrapped functionality as defined by user. While cloning a process, the users have now two options. Either they can choose to use the same system call vectors for the child which the parent is currently using or they can specify some other vector table while cloning. In case the child wants to use the same system calls that its parent is using the task_struct of the parent is copied to its child. Hence, the child will behave similar to its parent whenever it call a system call. If the users want to use some other set of system calls, they can specify it while cloning. We have made a new custom clone system call to support this.

III) Applications

The per-process system call can expose many useful applications to the user. We have tried to implement some of the use cases in our implementation. Few of the usages are listed below -

1) New system calls can be created to fancy and useful tasks. System calls to ensure security or logging/profiling can be created. For example we have created system calls which do not allow unlink or stat on protected files.
2) The idea can be used to ensure process level security inside the system. Depending on the security level of the user, we can block system calls for that particular user processes.
3) Modification in Kernel code can result in serious issues. By implementing a per-process system call with our approach avoids a lot of modifications in kernel code without changing the behavior of other applications on the system. Hence we get easy kernel portability as well.

IV) Features

1) We can write user defined functions to override the normal behavior of a system call and use them as normal system call. In the assignment we have implemented some of the overridden system call functions.
2) We can wrap the original system calls to perform some extra functionalities like logging/profiling without changing the original behavior of the system call.
3) The child process can have same or different set of system calls according to the input given by the user while cloning.
4) We can change the system vector which is set for a user process using IOCTL commands even when the process is running.
5) A new system call vector can be registered or a registered system call can be unregistered.
6) While cloning a child process we can decide whether to create the child process with the newly created system call vectors or the default system call vector.

V) Design

In the design part we discuss about how we achieved our functionality.Basically we have modules for everything.

Register and unregister module:

The first and foremost module is to register and unregister system call vectors which registers new modules and unregisters them.While registering system call vector we give each system call vector a unique number.

Ioctl module:

As we cannot access kernel data structures through user modules to set a vector for a process and to remove a syscall vector for a process or to change a syscall vector for a process we need a kernel module.We are implementing it using Ioctl.Ioctl uses the register and unregister module.So unless ioctl is removed we cannot unload register and unregister module.

Syscall vector module:

Each custom system call vector is implemented as a module .When a process uses a vector we increment the reference count and when process ends we decrement the reference count so that when a process is using a module it cannot be unloaded.As we know that we set vectors for process through ioctl we created a dummy function and made the custom syscall vector module use it. So that user cannot unload the ioctl module until the custom syscall vector is removed.As we are using register and unregister functions of register and unregister module unless the syscall vector is unloaded register and unregister module wont be unloaded. We are saving the custom vectors of system calls in linked list and each vector is a linked list of system calls.

System call interception:

We are handling only 64-bit syscall part.In entry_64.s we are intercepting the actual syscall and redirecting it our syscall in common.c and based on override flag we check if it overridden or wrapped and based on that we go for normal execution or skip the normal execution.

We are saving the vectors in a linked list using struct which has variables vector_id, address of the vector and the reference count. We are saving system calls of the vector in a linked list where each node as system call number and the function pointer of the system call.

VI) Implementation

We Implemented our design by modifying x86 architecture in linux 4.6 kernel. System calls are executed through two paths - slow path and fast path in linux which happens in entry_64.s. We directed every system call through slow path. Slow Path calls a callback function in common.c. In this function the system call is called with its corresponding system call number. Here we check whether the process has been assigned to a custom vector of system calls by checking the task_struct of the process where we saved the address of the custom vector using ioctl. If the process is assigned to a custom vector then we check whether the vector has a overrided or wrapped system call of the same system call in it. If so, we call it in place of original system call if the system call present in vector is overrided or else we call it and the original system call if the system call present in vector is wrapped. If the process calls a system call which is not present in the custom vector then we call the original system calls. Custom vectors have a list of system calls with function pointers for calling them. We call the overrided or wrapped system calls using these function pointers.

6.1 Module for registering vector

This Module Provides Functionality for adding/removing a custom vector of system calls. This has functions for adding or removing a overrided or wrapped system call to the the vector. Adding or removing a system call to the vector basically involves traversing the linked list and adding or removing a node with system call number and the function pointer to the node. This Module has also functions for getting the vector address when the vector id is given which will traverse the list of vectors and returns the corresponding vector address. It has function for reducing the reference count of a custom vector, this also traverses the list of vector and reduces the corresponding vector's reference. It also has a function for displaying the list of the vectors loaded on success it will write the list of vectors loaded to a file. We are using a mutex lock so that list of vectors cannot be accessed parallelly. We are using try module get and module put for managing reference count of the custom vectors. These custom vectors can be unloaded only when the reference count is zero i.e. when no process is using them.

6.2 Custom Vector of System Calls:

Custom Vector of System Calls is implemented as a linked list of system calls. Each system call has a system call number and the address of the system call. We created ten such system calls which can either override , wrap the original system call functionality.
The modified task struct contains vector id which will point to the vector id of the custom system call vector , it will also contain the address of the custom system call vector so that it

can access the system call vector.We implemented two custom system call vectors with five system calls in each system call vector.We illustrated in these system call vectors how these loadable system call vectors can be used for extra functionality. For example for processes like “httpd” we are denying the ability to make system calls like mkdir, rename and chdir.

6.3 Communicating via IOCTL

In order to communicate with register module we need some way. So we used a ioctl character device for this purpose. The purpose for communicating with register module is that we can set/remove a vector to a process. We created the characted device in /dev/ioctl_device with a major number 89 and minor number 1 which was chosen randomly. We register the device using the mknod command.

```
syntax : mknod /dev/ioctl_device c 89 1
```

The ioctl device has 4 ioctls: SET_IOCTL, REMOVE_IOCTL, LIST_ID, LIST_VECTOR. These are used for the following purposes:

SET_IOCTL - This ioctl is used by user process to set a custom vector to it. This module basically gets the corresponding address of the vector with the vector_id given and then adds vector_id and vector address to the task_struct of the process and also increases the reference count of the vector.

REMOVE_IOCTL - This ioctl is used by user process to remove the custom vector assigned to it. This module basically reduces the reference count of the vector and then removes the vector address and vector_id from task_struct of the process.

LIST_ID - This ioctl is used by user process to get the vector id assigned with a process. This module gets vector id from the task_struct of the process.

LIST_VECTOR - This ioctl is used by user process to get the list of the custom vectors loaded. This module basically gets the list of vectors by traversing the linked list of vectors and then writing them to a file.

6.4 Custom clone implementation:

We are providing two variations of clone. In the first one we can decide if the child can inherit the system call vector of the parent or it will get the default system call vector. For this we modified original clone system by adding a new flag. we have defined the new flag as CLONE_SYSCALLS in sched.h , we can signal through this flag to do_fork to decide if the child process gets the vector id of the parent or not.

In the second one we are creating a new system call (clone2) by adding a new entry in system calls table in x86 architecture. Then defining the new system call clone2 in fork.c. We have modified do_fork such that it takes a extra integer as the sixth argument. We pass the vector_id which has to be assigned to the child process through this extra argument. We set the vector_address and vector_id to the task_struct of the child. So the child process will start with the vector_id passed by user.

VII) How To Run:

Register Module - register_unregister.c and .h, override_syscall.h
Ioctl Module - ioctl.c and .h
Custom vectors - custom.c and custom_mkdir.c
Install_module.sh - script to make the above modules and test code.

To test the functionality of clone , three test files have been created:
./clone_default -> To test if default system call vector is passed to the child .
./clone_syscalls <vector_id> <Y/N> - To test clone_syscalls flag on/off passes the system vector of the parent to its child
./test_clone <vector id of parent process> <vec id of child process> - To set a different vector id to the child on clone.Parent will have vec id of parent and child will be assigned vec id of child. This is implemented with the clone2 system call which was created.

To test the functionality of System call vectors :
./test_vector2 <vector id> - the passed vector id will be set to the process created by running this.Printk messages will show that the process is executing system calls from the system vector which is set.
./test <vector-id> -to set first system call vector and to test it
./test_protected <vector-id> to set first system call vector to check the functionality for protected files.
./test_set_diff_vector <vector-id> In this there is a sleep time of 50sec and when a user wants to update a new syscall vector they can do it using uioctl

uioctl.c - User ioctl program for setting,removing the vectors, listing the vectors and vector_id when pid is given. This ioctl program is used for doing the above functions by separate process rather than the process setting its own vector.

Syntax to run:

```
./uioctl SET [Vector_ID] [P_ID]
./uioctl REMOVE [Vector_ID] [P_ID]
./uioctl VECTORS \n"
./uioctl VECTOR_ID [P_ID] \n"
```

VII) Limitations

(i) We are using ioctl for communicating with the register vector module. As said before we are using try module get for our custom vector modules to know the number of processes using them. This make sures that the custom vectors modules will be unloaded only when its reference count is zero. But in cases where processes are killed suddenly after assigning a custom vector to them. The reference count of the module will not be equal to zero. In such cases module can't be unloaded.

(ii) We made changes in x86 architecture so that overrided/wrapped system calls can be called in place of original system calls. We changed entry_64.s for this purpose. It had two paths for calling system calls - fast path and slow path. Fast path is used when system call has less arguments and slow path is used when there are more arguments. We directed every system call to slow path. So that we can deal with only one case. So this might affect the performance of the system.

VIII) References

1) Linux Cross Reference - http://lxr.free-electrons.com/source/kernel/?v=4.6