Linux Kernel
System to support per-process system call vector
Group CSE506P01

Shilpa Gupta 110948405
Tanwee Kausar 110937649
Salman Masood 110614711

{shilgupta, tkausar, smasood}@cs.stonybrook.edu

1) Introduction

System calls in the linux operating system are the way a user mode process can communicate with kernel mode. A user process can request the kernel to perform certain system calls on the behalf of user. This communication happen via interrupts where user processes put certain system call number and arguments in the registers and issue an interrupt. Kernel interrupt handler fetch the system call number from the register, consult the system call vector table in kernel and call the corresponding function with the arguments given in registers. All system call numbers are defined statically in kernel, these are defined in a system call vector/table. The implementation of these system calls are fixed and same for all the processes. Unlike BSD, linux does not allow dynamic overriding of these system call implementations. In this project we are trying to solve this problem and exploring a way by which a process can use its own implementation of system calls.

2) Problem Statement

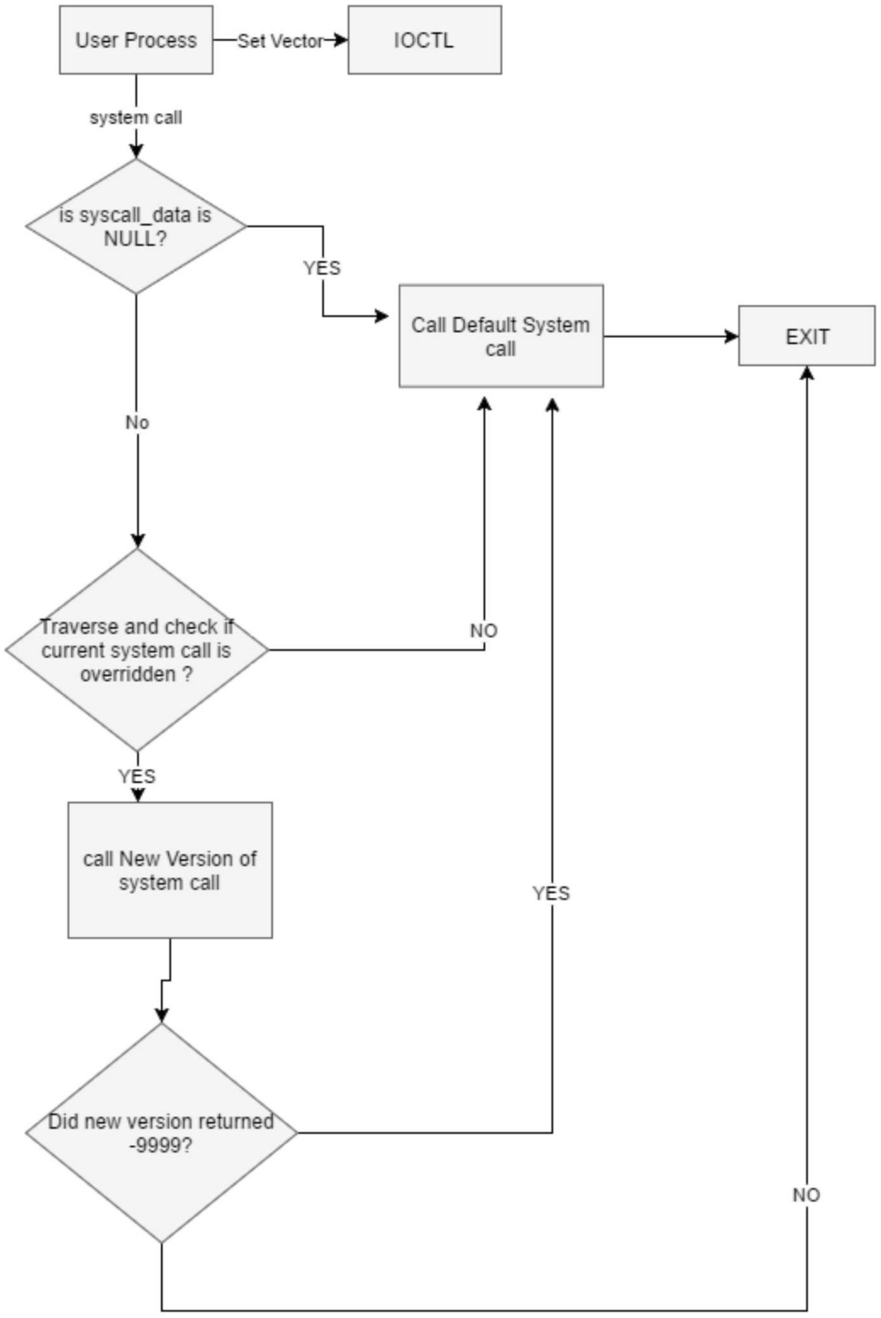
The processes are bound to use existing implementation of system calls in linux kernel. But there can be many useful scenarios where the implementation can be processes specific. For example from security perspective, certain process might want to allow read to a particular type of files only if the user is privileged. This check is not performed in existing read system call. To make sure fully that this check performed every time any user wants to access this file it is a better way to have a overridden version of read system call. We are attempting to develop a system where every process can override its system call vector with new implementation of system calls. Also a mechanism to make child process inherit the parent's system call vector address as well as to support a different vector id as an input which will be given to the inherited child.

3) Design

Since every process can have their own different system call vector, the first task was to associate the new vector with the process. We added new field in *task_struct* for this purpose. this field is a void* pointing to the syscall call vector which current process is using. We named this new field *syscall_data.* We also need to store the current registered system call vectors in the kernel some where so that any process can use one of the available system call vectors. We are using proc file system for this. Inside proc file system we created an file named syscall_vector which will contain all the current registered vectors. Every vector is represented by below structure

```
struct new_vector{
    char vector_name[MAX_VECTOR_NAME_LEN];
    unsigned long vector_address;
    int ref_count;
    struct module *vector_module;
    struct new_vector *next;
};
```

Every vector has a name and as we can see all the vectors will be linked to each other. vector_module is the kernel module which is implementing all the system call current vector has. We are also maintaining a reference count to each vector so that we can make sure that the module that is implementing system calls is not removed if the reference count is more than 0. Basically reference count represents number of processes using current vector, vector_address is the address of the list of system calls which current vector consist of. For setting and removing the system call vector of a certain process we are using ioctl. Other than this every vector will be implemented by a certain module which needs to be registered with the kernel. This module will be implementing the overridden version of the system calls. To implement the functionality in the kernel where the overridden version of the system call being called rather than the original one we have modified the assembly code in "entry_64.S". We have forced all the system calls to come via slow_path instead of fast_path. To call our modified version of system call we have make sure that before calling the original system call it should call our wrapper function. To achieve this we have modified part of code in common.c file which controls how a system call function is called in the kernel.
Below is descriptive Flow Diagram of our design.


User Process
Set Vector
IOCTL
system call
is syscall_data is NULL?
YES
Call Default System call
EXIT
No
Traverse and check if current system call is overridden ?
NO
YES
call New Version of system call
YES
Did new version returned -9999?
NO

We will explain the different modules and their working in features section.

4) Features

a. Registering a vector

To un/register vectors , we have written the module reg_sys_vec. This is the primary module which should be loaded before any vector module. This module provides framework to register any vector in the kernel. We are maintaining a linked list of vectors which are registered with the kernel. register_syscall() will add the new vector into the linked list. If a vector is not needed anymore it can be unregistered by this module. Normally when a new vector is added as a module it should register itself while loading the module and unregister while unloading.To unregister a vector, unregister_syscall method is called by the vector module that is to be removed from the kernel.

The vector module also takes care of the reference count of each vector so that we can refer to it while removing a vector module. At the time of registering a vector, its reference count is set to 0. Whenever, a process sets a vector for itself through ioctl, reference count is increased for the respective module. While unregistering, reg_sys_vec module checks if this reference count is zero or not. If it is zero, it returns 0 and allows the removal of module successfully otherwise , it returns negative value and doesn’t allow the removal of the module. The function get_vector_address is called whenever a new process requests to use a vector. This function increases the reference count of the respective vector by one. The function reduce_ref_count decreases the reference count by one whenever a process requests to stop using the vector through ioctl.

The reg_sys_vec module also creates an entry (syscall_vectors) in “/proc” and modifies its read file operation function to list the vectors currently registered on the kernel and number of processes using it. The function show_vectors traverses through the linked list of vectors and puts the vector name and number of processes using it (reference count) in the read buffer. Hence, whenever we cat proc/syscall_vectors , it displays information about our vectors.

b. Clone system call

We wanted to achieve a way by which we can specify to the clone system call that the cloned child needs to inherit parent process’s system call vector. We have modified current clone system call for this. We have added a support for new flag in clone system call called *CLONE_SYSCALLS. We ha*ve modified the kernel file fork.c and its function _do_fork() in which we added a check for this flag, if given then while copying the task structure we copy the syscall_data as well if not given we don’t copy the syscall_data and the child process get assigned default vector.

In addition to this we wanted to pass a vector to clone system call so that when the child process is created it get the new vector assigned to it. We have implemented and new system call for this called clone2 this system call in implemented as a kernel module which accepts an extra parameter *vector_name,* if passed then while creating a child the child process get assigned this vector rather than the default one. We have modified part of kernel code to achieve this, added a new parameter in *_do_fork()* method. Now the check inside _do_fork has become if the CLONE_SYSCALLS flag is given then copy the parent's vector to child else if the new vector name has been passed as a parameter assign this vector to the child else assign default vector to the child.

c. IOCTL

Any given user process needs some way of communicating with the existing vectors to use them or remove them. This we are achieving using by this module. For attaching these ioctl functionalities we are creating a character device in the kernel and registering it with the kernel at the time of loading of module. Any process wants to communicate with the vectors can open this device and pass the commands in order to perform certain actions. The commands that are supported by this ioctl are

1. IOCTL_SET_VECTOR : this command is passed with the vector_name argument, the purpose of this command is to set the process's vector to given vector. To achieve this we first increase the reference count of ioctl module by doing *try_module_get* then we try to get the address of the given system call vector if got, we assign this vector address to the process task_structure's syscall_data field. If failed we put the module back. We are using mutex lock to perform locking before any of the ioctl command executes.

2. IOCTL_REMOVE : this command is passed to ioctl module by a process when it no more wants to use this vector. This command is also passed along with a vector name argument which the process wants to get rid of. While removing the vector we first reduce the reference count of the vector, then we remove it from the task_struct of the process we also reduce the reference count of ioctl module by doing *module_put* , to make sure we reduce the reference count of of ioctl module which we increased while doing set vector ioctl command.

3. IOCTL_GET_VECTOR : we are using this command to display the vector given process is using. This command is called along with an argument which is the process id of the process of which we want to know the vector name. To achieve this we first extract the task_struct of given process fetch the vector address from it and find the corresponding vector name and return.

4. IOCTL_SET_DEFAULT: This ioctl call will be used to set the vector of any running process to default. This expects pid of the process as input and sets the syscall_data field of task structure to NULL after removing the old vector. Hence, assigning default vector the given process.

d. Obscenity vector

This vector will provide a feature to hide insensitive words from the user while reading, writing, creating file etc. It will also make sure that the file is not owned by the blacklisted users. For this the obscenity_filter module will store the pids of blocked user and a dictionary of obscene words. This vector will modify open(), create(), read(), write(), fchown() and close() system calls. Each will provide following feature:

a. open(): If any process will try to open a file with O_CREAT flag, which contains obscene words it will not allow it.

b. creat(): If any process will try to create a file, which contains obscene words it will not allow it.
c. read(): If a file contains obscene words, read function will replace the obscene word with '*'. For example if "dog" is considered obscene then the user will read "d**".
d. write(): This function will make sure that user will not write any obscene word into the file directly. Write of this module will replace the obscene words with '*' as described above.
e. fchown(): This function will not allow change of ownership of a file to any blocked user defined in this module.

e. Secure vector

This vector provides disabling of system calls when filenames contain “protect” keyword. This is an example to show that if some confidential file is present in our system, we can disable required system calls to protect those files.
This vector supports the following five system calls :

a. link()
b. unlink()
c. mkdir()
d. rmdir()
e. chmod()

When we install the module, it first adds all these system calls (system call number and respective function pointer) to itself. Once done, it registers itself in the vector linked list.
In all the above calls, we first check if the filename contains the keyword “protect” or not. If it doesn’t contain the keyword, this vector returns -9999 to indicate that we have to call the respective original system call. If it contains the keyword, it returns 1000 to indicate that we have to disable the original system call and prints the message that the user can’t operate on the file as it is protected. This functionality is useful for securing our confidential files, hence the name is secure_vector.

5) Files contained in this submission:

1. regvec - Contains code to un/register vectors.
2. obscenity_filter - Module to implement obscenity filter vector
3. secure_vector - Module to implement secure vector vector
4. syscall_clone - Module to modify original clone() system call and implement new clone2() system call
5. vec_ioctl - Module to implement required IOCTL
6. Test_demos - Folder which contains test files.
7. README.HW3 - This readme file.

Linux code changes:

8. “arch/x86/entry/entry_64.S”, “arch/x86/entry/common.c”
   Changed assembly code for calling our modified version of system call.

9. “arch/x86/entry/syscalls/syscall_64.tbl”, “fs/open.c ”, “include/linux/sched.h”, “include/linux/syscalls.h ”, “kernel/fork.c”
   Code changes for modified clone() system call and new clone2() system call.

6) Execution

Initial setup:

a) git clone
b) cd hw3-cse506p01/
c) cp kernel.config .config
d) Make
e) Make modules_install
f) Make install
g) Reboot
h) Cd /usr/src/hw3-cse506p01/hw3
i) ./mymake.sh (compiles and install the modules in proper order)
j) cat /proc/syscall_vectors (It will show the registered vectors)

Testing:

To make all the test user programs follow below initial steps

a. cd test_demos/
b. Make

Test case 1:

- ./test_obscenity_filter.o obscenity_filter
  Expected behaviour : This should not allow creation of file having obscene words either using open() or creat() system function call. If a file contains some obscene words defined in obscenity_filter module, then read() should replace the obscene words with ‘*’. Write() also should check the buffer before writing and write the modified buffer. fchown() function call should not allow to change the ownership of file to some black-listed users.

Test case 2:

- ./sleep_test_obscenity_filter.o obscenity_filter&
  Expected behaviour : This will put the test_obscenity_filter.o process to sleep, so that we can have a process running in background with our implemented vector.
- ps
  Expected behaviour: get the process id of previous process.

Test case 3 :

- ./get_proc_vector.o ‘pid’
  Expected behaviour: This will call IOCTL_GET_VECTOR ioctl call to get the vector name of process with process id as ‘pid’. If no special vector is assigned to it, output will be “default” vector.

Test case 4 :

- ./set_default_vector.o 'pid'
  Expected behaviour: This will use IOCTL_SET_DEFAULT ioctl call to set the vector of the given process to "default".

Test case 5 :

- ./get_proc_vector.o 'pid'
  Expected behaviour: Again check the vector name of the given process. It should be set to "default".

Test case 6 :

- ./test_secure_vector.o secure_vector
  Expected behaviour: Any file containing the 'protect' keyword in its filename won't be able to perform operations - link, unlink, mkdir, rmdir and chmod.

Test case 7 :

- ./test_clone2.o obscenity_filter
  Expected behaviour: In this user program we are making call to our own version of clone2 system call and passing obscenity_filter vector as argument which is resulting in the assignment of obscenity_vector to the child, both parent and child are performing read system call, since child has obscenity_vector assigned to it, it should print blurred version of text while parent should print normal text.

Test case 8 :

- ./test_clone_flag.o obscenity_filter
  Expected behaviour: Inside this user program we are setting our flag CLONE_SYSCALLS while calling clone system call which is causing same vector assignment to child. Since both parent and child are making read system call we should see the obscene words blurred in the output of both.

7) Conclusion

We have developed a linux-kernel based system to support customisation of system call for any process . We implemented this by creating system call vectors as loadable kernel module. These vectors include certain system calls that need to be customised for certain processes. We created two vectors 'obscenity_filter' and 'secure_vector'. 'Obscenity_filter' vector module provides the functionality to hide any improper/obscene context in a file . And 'secure_vector' module provides the functionality to disable operations for files containing protect keyword. To link any process with these vectors, we have leveraged the task struct's void* syscall_data member. We store vector address in the process' syscall_data when the process sets a certain vector to itself. We have used ioctl to allow this communication between the process and the kernel that which vector the process wants to use. The ioctl also allows to change the system call vector of a running process. We have modified clone() system call to support a new flag that tells whether the child process can inherit the parent's vector or not. Also, we have added new clone() system call to pass a vector to the system call so that when the child process is created, it gets the new vector assigned to it.

8) Reference

- http://lxr.free-electrons.com/
- https://github.com/abhishekShukla/System-Call-Inherit
- http://man7.org/linux/man-pages/man2/syscalls.2.html
- http://man7.org/linux/man-pages/man2/clone.2.html
- http://stackoverflow.com/questions/21004377/where-is-the-clone-system-call-define-in-the-linux-kernel
- http://stackoverflow.com/questions/18904292/is-it-true-that-fork-calls-clone-internally

9) Test cases:

Tests	Result
Prevent unloading of module when any process is using its vector	Pass
List all existing syscall vectors (and any additional useful info about them, such as how many processes use them).	Pass
List the syscall vector name of a given process	Pass
Default syscall vector for all processes, inherited to children	Pass
clone(2) CLONE_SYSCALLS flag support	Pass
un/loadable syscall vectors	Pass
ioctl(2) support: change syscall vectors	Pass
new version of clone(2) to start with different syscall vector	Pass
ioctl(2) support: list syscall vectors, vectors for PID, etc.	Pass
Two new syscall vectors other than default, with at least five syscalls each.	Pass