System Call Inherit in Linux Kernel

Team:
Prankur Gupta - 108492684
Abhishek Shukla Ravishankara - 107598884
Sumit Singh Bagga - 108235636

Objective:
To implement a scheme where a process and its child processes in Linux have an option to use a new/modified set of system calls.

Motivation:
System Call Inheritance can prove to be a powerful scheme and can have the following applications:
1. System calls can be modified to suit application specific usage of the OS. For Ex Logging, Profiling etc
2. Security Applications: Depending on the security level of the user, the set of system calls allowed can be different.
3. Additional functionality can be added to the existing system calls fairly easily, maintaining the portability of the kernel. Kernel capable of allowing a user to extend system calls without changing the behaviour
of other applications on the system is required.
4. BSD has the feature of overriding system call table at a process level thus allowing access to kernel data structures without having a risk of writing an unsecure or bug prone kernel code.
5. By this approach a user can extend system calls without changing behaviour of other applications on system.

Design:

What we really want is a system that can fulfill the criteria of an application(wanting a new set of system call) without enforcing some undesirable changes for the application/processes that do not need that additional functionality.

We would be designing a Loadable Kernel Module, through which a user can override the default sys_call_table with the system call vector desired by the application.

A new system call can be added which is a modified version of exec, having at least two more arguments, a user system call vector name and a flag to indicate if the process has to override or not.

```
int exec_override(const char *path, char *const argv[], char *const envp[],
                  const char *system_call_vector_name, … );
```

A structure to override a system call will possibly look like this:

```
struct syscall_override {
        unsigned int syscall_number; /*System Call Number*/
        void *fun_ptr;  /* This pointer pointer to the function that will override a system
call*/
        void *extra_data; /*Allowing further extension if needed later*/
};
```

This information should be made available by each process/ application. As this is per process requirement, so we need a data structure that is process specific, so we can use task_struct, which can store the address of the system call vector required by the process. Using this approach, we can also set up a check to distinguish between the processes which have overridden the system call vector from those who will follow the normal system call vector.

The entry point of the system calls needs to be changed to make necessary checks if the process uses a overridden system call. If it is supposed to use the overridden system call, to check if its valid. If not valid, we will be returning a proper error code back.

We need to store all the system call vector available to the process for overriding the default system call vector, so for this we would be maintaining the list of the registered system call vectors in the proc filesystem i.e. the name of the system call by which the process wants to override its default system call vector. We also might need another API for storing and removing these new system call vectors. We will also need get() and put() functions to increment and decrement the ref counts to the new system call vectors.

```
// list of registered syscall vectors that will be stored in filesystem.
struct syscall_vectors {
  int ref_count;
  struct custom_syscall_vector csv;
  struct syscall_vectors *next;
}
```

The child process formed due to fork(2) or clone(2) will also inherit the system call vector of the parent. The syscall vector that is overridden by processes,is provided as a kernel module. We should not allow the removing the system call vector from proc filesystem until all the processes(parents and their children) using that overridden

syscall vector have completed their execution. So for that we will maintain a reference counter for that vector.

User Code: An ioctl code to pass the address of new syscall_vector. Or a simple code to use the overridden exec syscall. And a program which calls system calls fork() or copy_process() which create a new task structure.

Creating a new exec like syscall to pass the address of overridden syscall_vector is a cleaner approach as this system call can be added to kernel statically and can be used for overriding any system call in future as a standard. Ioctl on other hand is simpler task to do and it is implementation dependant. So choice between ioctl and exec syscall has been delayed for now as a task to be done depending on time constraint.

Drawback: Since we are creating a new system call vector handler by overriding the syscall vector, a user can create a handler which allows a process to perform operations or tasks that it is not privileged to perform or may even cause the kernel to crash.

Demo:

1. We will choose a small set of system calls implemented in the native linux kernel, extend it, (probably with logging functionality) and create a new system call vector. We will also add a new system call and include it in the system call vector.
2. Create processes which fork/exec children processes and show that the new system call vector with both modified and new system calls was inherited.
3. We would be able to successfully unload the module only if all the dependent processes(child / parents) have successfully completed their execution.
4. If we are successful with the above all implementations, we will probably try to come up with a scenario in which system call inheritance can prove to a vital scheme for security applications and demo it.