Per Process System-Call table

Hw3-cse506p14

{alhussain, asarda, pkhemka, shppatil}@cs.stonybrook.edu

1. Abstract

Linux prohibits a loadable module from changing the global system call vector/table. Linux makes it hard to change system calls, either globally or on a per process (or process group) basis. Having different syscalls per process is useful for various scenarios. For example, to add tracing/printf messages for certain syscalls; to override the default behavior of a system call (e.g., to encrypt/decrypt file names); to prevent certain processes from invoking syscalls they shouldn't (e.g., a Web server typically has no business invoking mkdir(2), unless it was hijacked). We propose a solution for a more fine grained control over per process system call execution. Using IOCTL, api calls and modified version of clone, we allow addition, removal, assignment of new system call during runtime and blocking system calls.

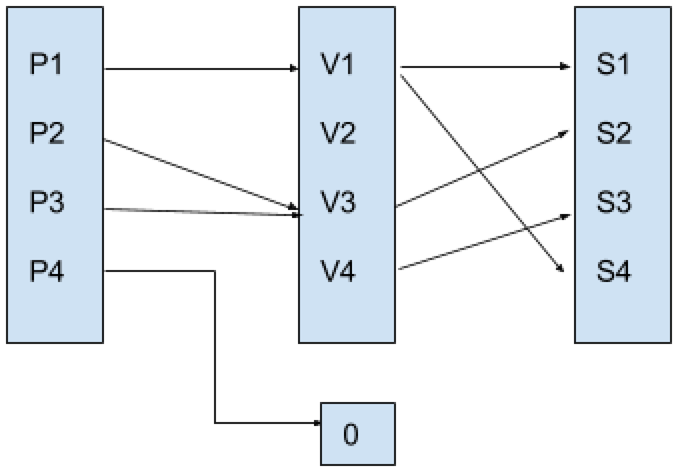
2. Design

By design, linux has codified system calls in kernel code, hence it is not easy to add/remove new system calls. Also, each process uses the same set of system calls identified by their system call numbers. For eg., system call number for “open” is 2. We provide a vector id for each process which is used to find a system call for that process. When a process makes a system call, it is intercepted and check is made to find if for a combination of vector id and a system call number, an overridden system call exists. If it does, the call is made, else default system call is used. If for a reason, system call number is blocked, ENOSYS is returned.

3. Data Structures

We have three sets of entries:

1. A set of system calls loaded and unloaded dynamically as modules. (S)
2. A set of vectors. (V)
3. A set of tasks (kernel maintained - processes/threads) (P)



3.1 System Calls

For every system call which is loaded as a module, we keep track of

- name (unique identifier),
- function address,
- module and,
- the system call number it represents.

```
struct syscall {
        char name[MAX_NAME_LENGTH];
        unsigned long fptr;
        short syscall_num;
        struct module * module;
} typedef syscall_t;
```

When the system call is assigned to a vector, its module reference count is increased. Similarly, when a system call is removed from the vector, its module reference count is decreased by one. This makes sure that, while system call is use, it cannot be removed from the system.

3.2 System Call Vectors

We maintain a list of system call vector. Each system call vector contains :

- Id (Identifier)
- List of overridden system calls
- Reference count
- Bitmap of blocked calls

```
struct syscall_vector {
        short id;
        short refcnt;
        char blocked[NUM_SYSCALLS];
        node_t syscall_num_list;
} typedef syscall_vector_t;
```

A process with vector zero(0), is mapped to default system call table provided as statically coded structure in Linux. For every non zero vector, we only keep track of the system call module that were added and mapped to this vector. If a system call number is not part of a vector, the corresponding default vector entry is used. This saves a lot of space, as most of the system call per vector will not be overridden. Also a vector which is already assigned to a process, will not be allowed to removed (reference counted), unless it has been released by the process or reassigned to any other vector.

Also, a bitmap of blocked system call is maintained per vector. Any system call which is blocked, will be rewarded with ENOSYS.

3.3 Linked List structure

```
struct node {
        struct list_head entry;
        void * data;
} typedef node_t;
```

Kernel linked list, unlike a regular linked list, has pointers to next and previous node embodied in the data itself. This particular design is not suited for many to many mapping scenarios. Hence, we defined a generic node structure with fields :

- List head
- Void * data (for any data type)

While creating a node, we used the concept of container, where the size of malloced item is the size of node plus the size of data. The data points to the pointer returned by malloc plus the size of the node. This improves cache locality immensely.

4. System Call Interception

We have intercepted system call at _do_syscall_64 function defined at arch/x86/entry/common.c. Here, using the vector id of the process, we make a call to the function to check if the system call is overridden. The corresponding function - any loaded system call or default system call is executed and returned. If the system call is blocked, ENOSYS is returned.

```
fn = check_syscall_override(nr & __SYSCALL_MASK);
```

5. IOCTL

We have implemented multiple ioctls(2) that can change the syscall vector used by a running process (PID). The ioctls can change the syscalls executed by a running process multiple time. We have also ensured that we prevent unloading of modules whose syscalls are in use.

The other IOCTLs - List all existing syscall vectors and number of processes that use them and another IOCTL which lists the syscall vector ID of a given process.

IOCTL functionality is defined as a loadable-unloadable module.

5.1 Location of files:

The Kernel side code for IOCTL is located in /hw3-cse506p14/hw3/syscall_module and user level code for IOCTL is written in : /hw3-cse506p14/hw3/ioctl_module.

5.2 Details of each IOCTL:

IOCTL NAME	PURPOSE
IOCTL_SYSCALL_TBL_ADD_VECTOR	Adds a vector ID
IOCTL_SYSCALL_TBL_REMOVE_VECTOR	Removes a vector ID
IOCTL_SYSCALL_TBL_ADD_SYSCALL_TO_VECTOR	Adds a syscall to a vector id
IOCTL_SYSCALL_TBL_REMOVE_SYSCALL_FROM_VECTOR	Removes a syscall from a vector id
IOCTL_SYSCALL_TBL_ASSIGN_VECTOR_TO_PROCESS	Assigns a vector to process
IOCTL_SYSCALL_TBL_GET_VECTOR_ID_OF_PROCESS	Gets the vector id of a process
IOCTL_SYSCALL_TBL_GET_INFO	Lists all existing syscall vectors and number of processes that use them
IOCTL_SYSCALL_TBL_BLOCK	A bitmap of blocked system call is maintained per vector. Any system call which is blocked, will be rewarded with ENOSYS. This IOCTL is used to disable system calls.
IOCTL_SYSCALL_TBL_UNBLOCK	This IOCTL removes disabled system calls.

5.3 Compiling Kernel Side Code:

Kernel side contains two files - hw3_ioctl.c and hw3_ioctl.h pertaining to IOCTL files in the syscall_module directory.

To compile the Kernel side code, the user needs to execute the command make from the syscall_module directory.

On successful compilation, one needs to load the ioctl module by executing the commands :

```
insmod syscall_tbl_mod.ko
insmod hw3_ioctl.ko
```

5.4 Compiling User Side Code and creating device file:

In addition to Kernel level code, we have implemented a user level code to test out these IOCTLs. The user level file is located in /hw3-cse506p14/hw3/ioctl_module.

It contains three files - user_ioctl.c, user_ioctl.h and Makefile

user_ioctl.c - contains the user level program to test out all IOCTLs. To compile user_ioctl.c, the user needs to execute the command - make.

On successful compilation, the user needs to execute this command -

```
mknod /dev/hw3_ioctl_test_device c 229 121
```

This command creates a device in /dev with major number 229. It can be viewed by executing the command

```
vi /dev/hw3_ioctl_test_device
```

This is because before using the ioctl system call , a device file needs to be created in /dev file system.

229 is our registered Major number and 121 is minor number. Both 229 and 121 were chosen randomly while ensuring that they don't conflict with any other numbers.

5.5 Executing User level program :

./user_ioctl [-c -p] [arguments]

It contains two flags

-c : for changing system calls. In addition to the -c flag, user needs to provide a default value which corresponds to the type of syscall manipulation that the user intends. It is explained below in the -c flags section.
-p : get vector id of process
Without any flags - list all existing syscall vectors.

Without any flags:

./user_ioctl : Lists all existing syscall vectors and number of processes that use them

Usage of -p flag:

./user_ioctl -p processid : gets vector id of process

Usage of -c flag:

./user_ioctl -c value vectorid processid

./user_ioctl -c 1 : add syscall vector
./user_ioctl -c 2 vectorid : remove syscall vector
./user_ioctl -c 4 vectorid syscallname: add syscall to vector
./user_ioctl -c 8 vectorid syscallname : remove syscall from vector
./user_ioctl -c 16 vectorid processid: assign vector to process
./user_ioctl -c 32 vectorid syscallnum: block syscall in vector
./user_ioctl -c 64 vectorid syscallnum: unblock syscall in vector

6. Process Life cycle - CLONE2

We have implemented the modified clone functionality needed for Per-process System-call table using a separate loadable module rather than modifying the clone system call which is part of the kernel. There are two types of scenarios covered:

1. Using CLONE_SYSCALLS flags. When using this flag as part of clone/clone2 system call, the child inherits the vector id from parent.
2. When using clone2 and without CLONE_SYSCALLS flags, assigns the vector provided as part of the function call.

When a process spawns, if it has a non-zero vector id, reference count for the vector id is incremented by one, similarly when a process exits, reference count is decremented by one.

```
asmlinkage long clone2( unsigned long   clone_flags,
                unsigned long   newsp,
                int __user      *parent_tidptr,
                int __user      *child_tidptr,
                unsigned long   tls_val,
                short int       vector_id
```

7. Testing

7.1 Unit Testing

The following api provided by lower level data structure application, were unit tested for different success/failure scenarios. It is available at /usr/src/hw3-cse506p14/hw3/syscall_module/test.c

```
int register_syscall(char * name, unsigned long fptr, short syscall_num, struct module * module);
int unregister_syscall(char * name);
int add_syscall_vector(void);
int remove_syscall_vector(short vector_id);
int add_syscall_to_vector(short vector_id, char * name);
int remove_syscall_from_vector(short vector_id, char * name);
int show_syscall_tbl(void);
int get_vector_by_pid(pid_t pid);
int assign_syscall_vector(short vector_id, pid_t pid);
int unassign_syscall_vector(pid_t pid);
int reassign_syscall_vector(short vector_id, pid_t pid);
int block_syscall_num(short vector_id, short syscall_num);
int unblock_syscall_num(short vector_id, short syscall_num);
unsigned long get_syscall_fn(short vector_id, short syscall_num);
int get_syscall_tbl_info(syscall_tbl_info_t * syscall_info);
int show_syscall_tbl_test(syscall_tbl_info_t * syscall_info);
```

7.2 Integration Testing

Different type of scenarios were tested.

1. Clone2 system call: Assigns a vector id to a new process spawned using clone.
2. Adding, Removing System call vectors: Adding/Removing new loadable system call vectors. Nine new system calls were created for testing purposes.
3. Adding/Removing system call vectors: Adding/Removing new system call vectors and assigning system calls to it dynamically.
4. Overriding system calls: Cloning process to execute system calls that were overridden. We have overridden system calls like open, read, write etc.
5. Ioctl: A script which tests each IOCTL call.