PER-PROCESS SYSTEM CALL VECTOR (PPSV)

CSE 506: Operating Systems | HW3 | Group P02
Bhushan Shah | bhushah@cs.stonybrook.edu
Mukul Sharma | muksharma@cs.stonybrook.edu
Prasoon Rai | prrai@cs.stonybrook.edu
Shivanshu Goswami | sgoswami@cs.stonybrook.edu

Introduction

System calls in the Linux kernel are defined within the kernel code. Alterations to these calls are not allowed via external mechanisms such as loadable kernel modules. There are, however, legitimate use cases in security which demand some measure of control as well as extensibility on the system calls.

We have designed a mechanism to introduce multiple independent sets of system calls, hereby called ***sys_vectors*** into the kernel as loadable kernel modules. Support for these vectors is built into the kernel and a developer merely needs to write the code for the customized system calls that he or she wants to extend.

These *sys_vectors*, once loaded, can be associated with processes and those processes in turn start using the customized versions of system calls. This mechanism also allows developers to block certain system calls, and to use the native system calls in the *sys_vectors* if required.

Design

SYS_XCLONE

Since, we wanted to keep minimum interference with base kernel, we have introduced a new system call ***sys_xclone*** which has the following features:
- It is similar to vanilla fork (for simplicity).
- It creates child processes and associates them to the *sys_vector* ID passes as argument.
- It does other minor tasks like getting a *refcount* on the *sys_vector*.

Note: We preferred to put this system call in the kernel itself instead of creating a new module.
The reason was to avoid exporting internal kernel functions that our system call uses. That would have been a security risk as well, as rootkits could in principle have exploited these exported inner functionalities.

SYS_VECTOR

Each new process created with *sys_xclone()* is assigned a ***sys_vector*** based on the vector ID passed to sys_xclone(). sys_vectors are defined in loadable kernel modules which have some custom system calls. The *sys_vector* contains a custom system call table similar to kernel resident *syscall_table*. The difference is that we put our own custom system call methods in the table, block some system calls as well as use some calls from the kernel resident syscall_table.

If we define a new custom system call *mkdir*, then sys table will look like this:

```
systabl[base_systabl + __NR_mkdir] = (unsigned long) custom_mkdir
```

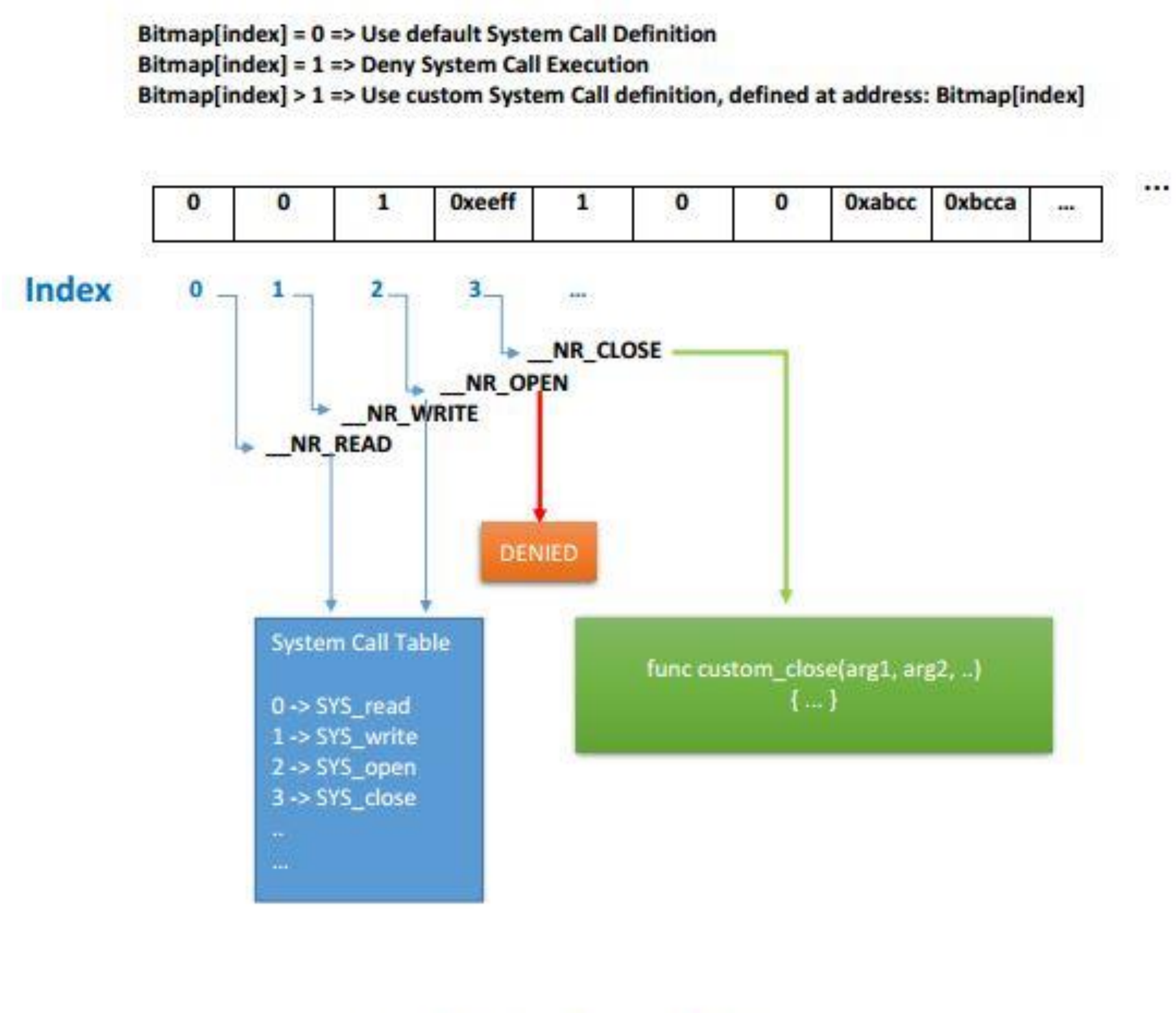
In addition to having new system calls, the table can accommodate default system calls and also restrict some set of default system calls i.e. deny them.

For default case:

```
systabl[base_systabl + __NR_mkdir] = CALL_DEFAULT
```

For deny case:

```
systabl[base_systabl + __NR_mkdir] = CALL_DENY
```



Bitmap[index] = 0 => Use default System Call Definition
Bitmap[index] = 1 => Deny System Call Execution
Bitmap[index] > 1 => Use custom System Call definition, defined at address: Bitmap[index]
0
0
1
0xeeff
1
0
0
0xabcc
0xbcca
...
...
Index
0
1
2
3
...
__NR_CLOSE
__NR_OPEN
__NR_WRITE
__NR_READ
DENIED
System Call Table
0 -> SYS_read
1 -> SYS_write
2 -> SYS_open
3 -> SYS_close
func custom_close(arg1, arg2, ..)
{ ... }

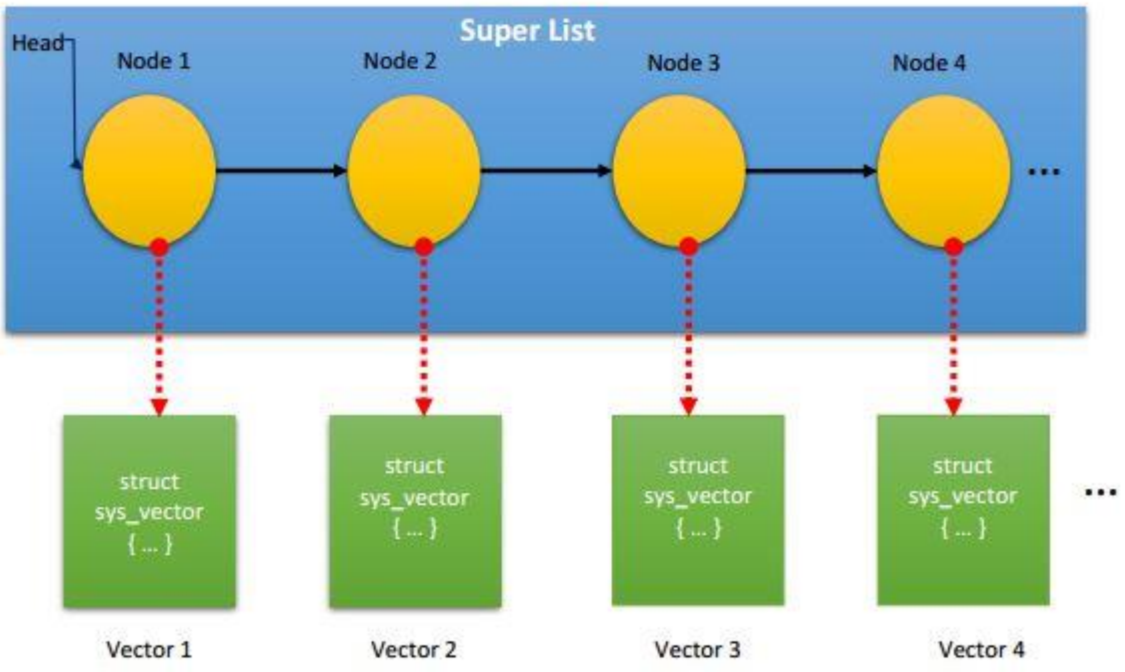
Sample sys call bitmap for a vector

sys_vector also contains ***refcount*** to track how many processes are using it. This tracking is useful during the removal of *sys_vectors* from the system. Since we want a safe exit mechanism, we don't allow removal of a *sys_vector* from the system if any process has a reference to it i.e. using/used its custom system calls.

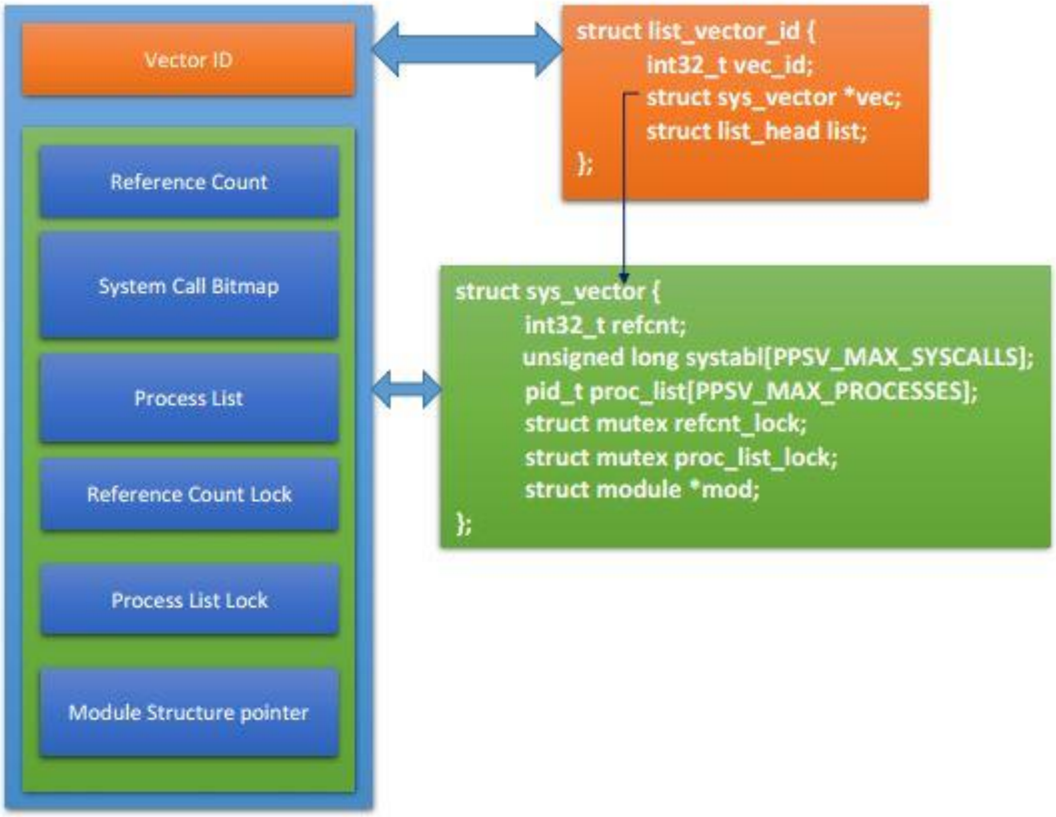
sys_vector also contains a list *proc_list* which keeps the *pid* of all the processes that have a reference to the *sys_vector*. In addition, there are **two mutex locks** for synchronization in *refcount* increment/decrement and *proc_list* update separately.

And finally, *sys_vector* contains the pointer to the ***struct module*** from which it is initialized. This is helpful while uninstalling the module safely.

We have modified the kernel to natively maintains a list that we call the ***super_list.*** This is a list of sys_vectors presently loaded in the system. More information on that is following.


Super List
Head
Node 1
Node 2
Node 3
Node 4
...
struct sys_vector { ... }
struct sys_vector { ... }
struct sys_vector { ... }
struct sys_vector { ... }
...
Vector 1
Vector 2
Vector 3
Vector 4

Global Super List: Each node points to a unique vector structure



```
struct list_vector_id {
        int32_t vec_id;
        struct sys_vector *vec;
        struct list_head list;
};
```

```
struct sys_vector {
        int32_t refcnt;
        unsigned long systabl[PPSV_MAX_SYSCALLS];
        pid_t proc_list[PPSV_MAX_PROCESSES];
        struct mutex refcnt_lock;
        struct mutex proc_list_lock;
        struct module *mod;
};
```

Vector structure: Various fields of the struct sys_vector

SUPER_LIST

In PPSV system design, we focused on the scalability. Developer can define as many custom system calls as required and pack them into *sys_vectors*. All he or she needs to do is to write new modules. For proof of concept, we have defined three such modules, each of which have a separate set of customized system calls.

To achieve scalability, we have defined ***super_list*****, a global linked list** which contains *sys_vector* and its *vector_id*. Every new *sys_vector* is registered to this list. Whenever we uninstall the associated module from the system, the *sys_vector* is unregistered i.e. remove from the list. Registration involves some memory allocation and lock initialization while deregistration involves the cleanup of the memory allocation done during the registration.

For synchronization across multiple un/install of different modules of *sys_vectors* and fetching the vector from the *super_list* while assigning it to the process, we have one **global lock** which protects the *super_list* from the corruption.

Workflow

Create process

To create a new process possessing a *sys_vector*, userland code calls our system call (sys_xclone) with a valid *vector_id*. *sys_xclone* first checks the validity of *vector_id*. Validation checks include whether the id is a valid number or not and whether it is registered into the system i.e. super list.

If validation is successful, a new process is created similar to vanilla fork behavior and the process is assigned the *vector_id* and the associated *sys_vector*. *refcount* is incremented on the *sys_vector* as well as on the module that initialized the vector.

Translation to custom call

When this process calls a system call, a number of steps happen. After the INT80 interrupt, the syscall handler --which is written in assembly and described differently for different architectures-- comes into picture.

There are two paths a system call can take from interrupt handler code: a *fast_path* and a *slow_path*. The *fast_path* is a direct call to the corresponding address in kernel native *syscall_table*. This is taken by calls that are not IO bound like *gettimeofday()*. Its code is in assembly. The slow path does some more checks and it goes through *do_syscall_64()* which is C code. For the purposes of this demo, we have ensured that the kernel takes a single path through *do_syscall_64()*.

Our hook lies in *do_syscall_64()* code. It checks the *task_struct* of the calling process. The process which uses our *sys_vector*s stores the address of the *sys_vector* it uses in its *task_struct*. We use this saved address and the corresponding system call NR number to call our method directly from the entry point.

After the call completes, we set its return value in AX register. Our custom system calls can also send return values indicating that after their execution, the original kernel resident call should also be made.

Exit process

When a process is done, we decrement the *refcount* on vector as well as on the module.

sys_vector modules

For proof of concept, we have introduced three modules named as *sys_vector1*, *sys_vector2*, and *sys_vector3*.

sys_vector1 has following custom system calls:

- *vec_1_open*
- *vec_1_creat*
- *vec_1_chdir*
- *vec_1_fchdir*
- **mkdir* => denied system call
- **rmdir* => denied system call

Rest are default system calls.

sys_vector2 has following custom system calls:

- *vec_2_mkdir_httpd*
- *vec_2_unlink* => avoid unlinking of protected files
- *vec_2 _kill* => deny to send signal to parent/sibling processes.
- **chdir* => denied system call
- **rmdir* => denied system call

Rest are default system calls.

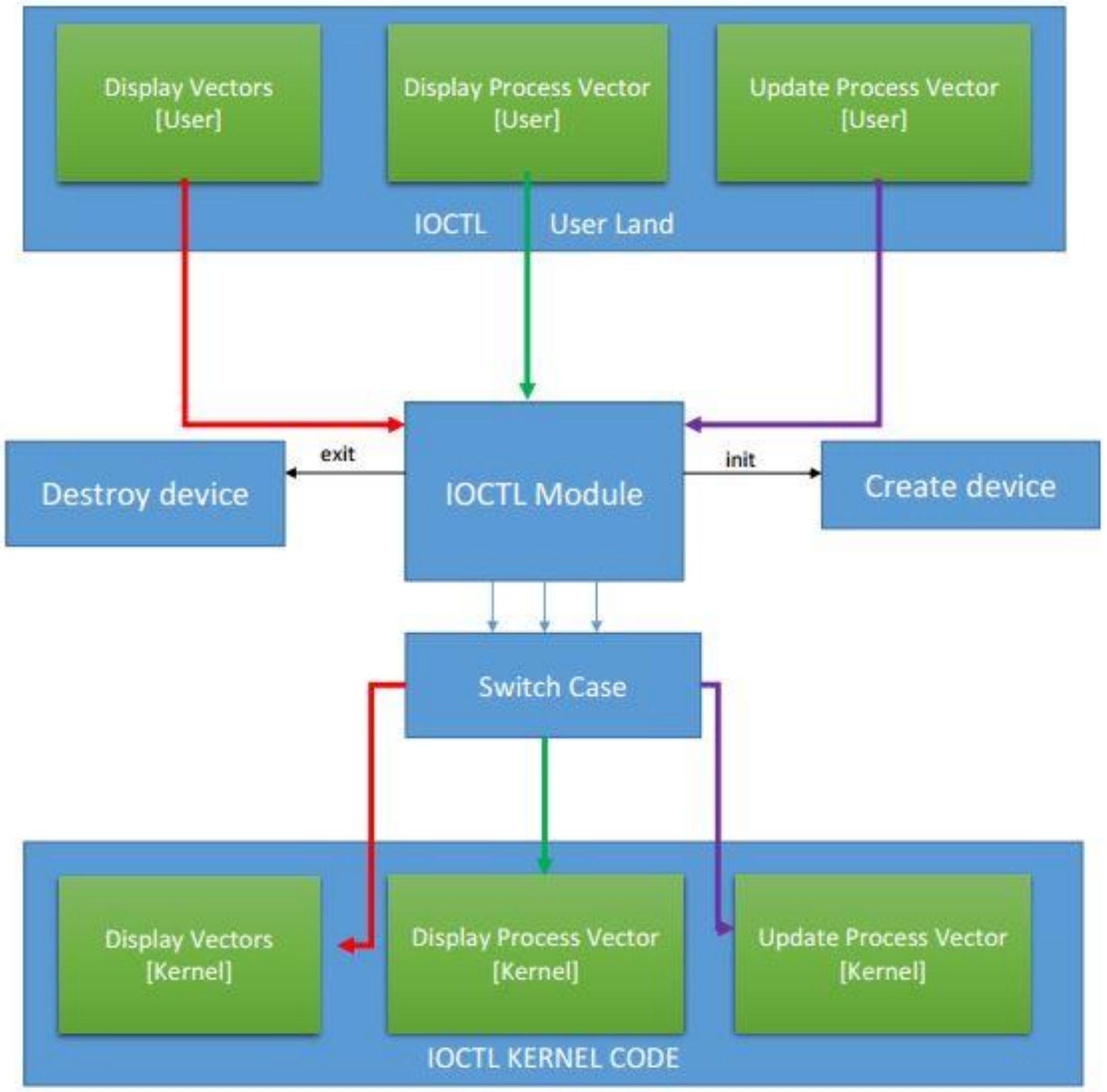
sys_vector3 has following custom system calls:

- *trace* => logs info about the following file operations
  - *vec_3_write*
  - *vec_3_link*
  - *vec_3_symlink*
  - *vec_3_rmdir*
  - *vec_3_unlink*
- **mkdir* => denied system call
- **read* => denied system call

Rest are default system calls.

Ioctl module

In order to issue commands to the kernel from user space, we have used an ioctl mechanism. We have implemented an *ioctl(2)* in a kernel module, and created multiple user programs which call this ioctl, each corresponding to one ioctl command.



IOCTL MODULE AND FEATURES

The following are the relevant files for this mechanism.

vec_ioctl.h

This is a header file which defines the commands to be used in the ioctl. Definitions of structs to hold command parameters are also in this file. The following commands are defined in this file:

VEC_IOCTL_DISPLAY_VEC_INFO
This command is used to display the following information about all the syscall vectors.

- vector id
- ref count
- which default syscalls are denied
- which custom syscalls are present
- which processes are subscribed to this vector.

VEC_IOCTL_DISPLAY_PROCESS_VEC
This command is used to display the aforementioned information, but only for the vector used by the specified process, taken as a parameter.

VEC_IOCTL_UPDATE_PROCESS_VEC
This command is used to update the syscall vector of a running process with another vector. The process id and the vector id are taken as parameters.

Apart from this, the structs to hold information about vectors are also defined in this file.

vec_ioctl.c

This is a kernel module which implements the ioctl. It contains a function *vec_ioctl* that handles the ioctl requests issued by the user program, and depending on the command issued, calls the corresponding function, also defined in this file, described below.

populate_vec_info
This function takes a struct as an argument, and populates it with the vector information. It is called when the user program issues a *VEC_IOCTL_DISPLAY_VEC_INFO* command.

populate_process_vec_info
This function takes a process id and a struct as an argument, and populates the struct with the information about the vector used by this process. It is called when the user program issues a *VEC_IOCTL_DISPLAY_PROCESS_VEC* command.

update_process_vector
This function takes a process id and a vector id, and updates the vector used by the process. It is called when the user program issues a *VEC_IOCTL_UPDATE_PROCESS_VEC* command. In the ioctl code, we generate a *SIGSTOP* signal for the process. This puts the process in suspend state. We then do some prechecks and update the process vector. Finally, we issue a *SIGCONT* and the process goes to runnable states again.

display_vectors.c

This file contains a user program to issue a *VEC_IOCTL_DISPLAY_VEC_INFO* ioctl command, and print the information in a human readable format.

Since the ioctl only returns the syscall numbers, but we want to print the syscall names, we have used **X Macros** to initialize an array of syscall names, and print the proper names corresponding to the numbers.

Note: Usage of X Macros causes some unavoidable errors while running *checkpatch.pl*.

display_process_vec.c

This file contains a user program to issue a *VEC_IOCTL_DISPLAY_PROCESS_VEC* ioctl command, and print the information in a human readable format.

update_process_vec.c

This file contains a user program to issue a VEC_IOCTL_UPDATE_PROCESS_VEC ioctl command.