ProcTrac: Linux Kernel Module to Monitor and Track File Access

B. S. Hong

December 26, 2018

Abstract

A Linux kernel module was built to keep a list of files to keep track of and raise/log an alert when a file in that list was accessed. In this project, this was accomplished by hooking into some system calls to raise an alert.

1 Design

The design of this project is fairly simple and straightforward; on one end of the program, is an interface with the user via some means. Following the UNIX philosophy of "everything is a file," it made sense to expose a file where the kernel module would keep its configuration (the list of files it would be monitoring for). On the other end, a handful of system calls would be hooked and whenever a syscall like *open(3)* would be called, it would check the parameters against the list and raise an alert when there was a match.

2 Challenges

Despite its simpleness, there were quite a few challenges that I've ran into. Most of the problem arises from writing a Linux kernel itself and often the complexity of doing so. There are many tutorials online that show how to write a basic kernel module. This was fine, but as I tried to add functionality to the module, it proved very hard for several reasons. Firstly, there weren't as many guides or documentations on the exact subsystem or the exact function that I was looking for. To make that worse, the Linux kernel went through a major internal change on the 2.6 release. The entire linking system changed and moved from userspace to kernel space [?]. This caused half the guides to become outdated since the kernel symbols were no longer exported and even if you find the syscall table, it is also marked read-only.

Another challenge was that it was just simply too different of an environment to learn to code for, especially if not well familiar with the kernel internals.

After a bit of research, I found out about ftrace. Ftrace is a tracing utility directly built into the kernel. Using ftrace, I was able to install the function hooks. One of the issues that I had while coding this part is that debugging was very annoying; normally, there's a bug and your program just segfaults, terminates, or spits out and error. In the kernel space, when something goes wrong, the kernel would panic and halt. I would then have to perform a hard reset on my machine and also lose all the debugging info.

Another really hard part was the user input part. Initially, I tried to create a config file like '*/etc/ptrac.conf*'. The users would modify that file and when the kernel module gets loaded, it would read from that file. However, this proved to be harder than expected. To be fair, opening up files from the kernel is considered a really bad practice anyways, and it was rightfully difficult in attempts to dissuade bad coders to take that route. If you wish to, the commit history would show that I've written a bunch of helper functions like *open*, *write*, *read*, *close* to try to make this work, but could not get read/write to work correctly.

The next thing attempted was to expose a *sysfs* file. The *sysfs* is a virtual filesystem provided by the kernel, very much like the */proc* or */dev* directories in a typical Linux system, that does not map to a real filesystem, but maps to the kernel memory, somehow. It basically exposes the kernel kobject models to userspace via a virtual filesystem. This part also had very little standard documentation and the guides that were found all used different functions and stuff. Eventually, I went through the kernel docs and was able to find and use the right functions to create a sysfs entry that the user and the module can communicate through.

3 Implementation

As described above in the previous section, the two main parts of the project are the hooking and the sysfs entry. Firstly, when the module is inserted, it creates a kobject and sysfs directory */sys/ptrac* (line 311). It then creates an 'attribute' of that kobject just created, pointing to the two handler functions that would read and write from the sysfs file */sys/ptrac/filelist* (lines 302,314). '__ATTR_RW()' is a C macro included from *linux/sysfs.h*. If inspected carefully, you may notice that *filelist_store* actually reads a string for filename and an int for access. This will be further discussed in the next section.

There actually is a third part of the module which is internal, and that's the linked list that stores all the filepaths. After being read, the filename is then stored in the *struct st_fcontrl*, which is a node in a singly-linked list pointed to by *flist*. The files are added onto this list through the function *filelist_store* and removed in either *filelist_store* when the access is 0 or when the kernel module is unloaded. The function *filelist_show* handles when a user process reads out the sysfs file; it simply traverses through the linked list and prints the access value and the full filepath.

After the sysfs entry is setup, the module installs all the hooks defined in lines 231 to 235. Additional functions can be added by adding another of these lines and creating *real_sys_name* and *hook_sys_name* like I have with other functions. They also have to be installed and removed. The functions *install_hook*, *ftrace_hook*, *register_ftrace_function*, *remove_hook*, *resolve_hook_address*, and *ftrace_thunk* are all either wrapper or helper functions that actually deal with the hooks.

In *resolve_hook_address*, there is a call to *kallsyms_lookup_name*. Since the kernel symbols are no longer statically exported, this is how the function addresses are dynamically resolved from its names.

The hooks, when called, all do very similar things: they first check if the filename provided is in the list of filenames to be monitored. If it is, then it prints an alert with the PID of the process that is trying to access the file. It then calls the original, real syscall and returns the returned value. One thing I figured out working on this part was that the decl spec *__user* actually meant something. It meant that the data was in userspace, and that it should not be dereferenced directly. That is why the call to *strncpy_from_user()* was necessary in *dup_fn*.

Things were working fine, until I tested hooking *sys_execve*. I realized that the filename would be sometimes a relative path rather than a full path. That's what the function *resolve_path* was meant to fix. It would get the full filepath based on the current working directory of the current process. The easiest method to accomplish this seemed to be to just call the syscall *sys_getcwd* after finding out that calling syscalls are allowed while in kernel space. Once again, *kallsyms_lookup_names* was used to try to resolve *sys_getcwd*, but the function just didn't work right. (I forgot to remove that line with *sys_getcwd*.) Anyways, the *resolve_path* function returns the full path in a new buffer, making sure to free all the old and intermediate buffers. (Another example of annoyance of using kernel functions: the usage for the function *d_path* was in the kernel source and had a "NOTE" which was actually pretty significant in its usage).

4 Future Possibilities

There were a few more things I wanted to add and a few more things that I could've added, but I'll list and explain them here.

The previously mentioned 'access' variable that gets read and stored along with the filename was supposed to be like the flags parameter. Different access values would keep track of different actions for that file. For example, access value of 1 may only keep track of *open()* syscalls, or 2 may only keep track of removal of that file (*unlink()* and *unlinkat()*), and so on.

Something I started to deal with but didn't quite cover everything was relative paths. In *resolve_path* it removes all the leading './'s but nothing else. So anything that begins with a '../' or any dot/dot-dot directories in the middle would not be handled correctly and likely will not match any filenames during the search.

Another possible feature of this module could be to actually not only monitor, but also deny certain actions. Upon looking up the filename and comparing the access values with the attempted action, the function hook can return an error value instead of calling the real syscall.

Appendix A Source Code

The full project, including the Makefile, can be found on my GitHub page https://github.com/awkwardbunny/proctrac

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kobject.h>
#include <linux/slab.h>
#include <linux/ftrace.h>
#include <linux/uaccess.h>
#include <linux/fs_struct.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Brian Hong");
MODULE_DESCRIPTION("Process File Access Tracker Kernel Module");
MODULE_VERSION("0.1");

// Helper function to copy string from userspace
static char *dup_fn(const char __user *filename){
  char *kfn;
  kfn = kmalloc(512, GFP_KERNEL);
  if(!kfn)
    return NULL;
  if(strncpy_from_user(kfn, filename, 512) < 0){
    kfree(kfn);
    return NULL;
  }
  return kfn;
}
```

```
static long (*sys_getcwd)(char __user *buf, unsigned long size);
static char *resolve_path(char *fn){
  char *buf, *buf2;
  char *cwd;
  int len;
  int offset = 0;

  // No need
  if(fn[0] == '/')
    return fn;

  while(fn[offset] == '.'){
    if(fn[offset+1] == '.' && fn[offset+2] == '/')
      //offset += 3;
      // TODO: Remove top path
      continue;
    else if(fn[offset+1] == '/')
      offset += 2;
    else
      break;
  }

  // Get cwd
  buf = kmalloc(1024, GFP_KERNEL);
  if(!buf) return NULL;
  cwd = d_path(&(current->fs->pwd), buf, 1024);
  len = strlen(cwd);

  // Concatenate two halves
  buf2 = kmalloc(1024, GFP_KERNEL);
  strcpy(buf2, cwd); // cwd
  buf2[len] = '/'; // '/'
  strcpy(buf2+len+1, fn+offset); // filename

  kfree(buf);
  kfree(fn);
  return buf2;
}

// Definition of data structures to keep track of files
typedef struct st_fcontrl fcontrl;
typedef struct st_fcontrl {
  char fn[512];
  int access;
```

```c
  fcontrl *next;
} fcontrl;
fcontrl *flist = NULL;

// Helper function to search through file linked list
static int search_flist(char *filename){
  fcontrl *fcp = flist;
  while(fcp){
    if(!strcmp(fcp->fn, filename))
      return fcp->access;
    fcp = fcp->next;
  }
  return 0;
}

// Definitions and functions for hooking
struct ftrace_hook {
  const char *name;
  void *function;
  void *original;

  unsigned long address;
  struct ftrace_ops ops;
};

#define HOOK(_name) \
  { \
    .name = #_name, \
    .function = hook_##_name, \
    .original = &real_##_name \
  }

static int resolve_hook_address(struct ftrace_hook *hook){
  hook->address = kallsyms_lookup_name(hook->name);
  if(!hook->address){
    printk("unresolved symbol: %s\n", hook->name);
    return -ENOENT;
  }

  *((unsigned long *) hook->original) = hook->address;
  return 0;
}

static void notrace ftrace_thunk(unsigned long ip, unsigned long parent_ip,
    struct ftrace_ops *ops, struct pt_regs *regs){
```

```
  struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);
  if(!within_module(parent_ip, THIS_MODULE))
    regs->ip = (unsigned long) hook->function;
}

int install_hook (struct ftrace_hook *hook){
  int err;
  err = resolve_hook_address(hook);
  if(err)
    return err;
  hook->ops.func = ftrace_thunk;
  hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_IPMODIFY;

  err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
  if(err){
    printk("ftrace_set_filter_ip() failed: %d\n", err);
    return err;
  }

  err = register_ftrace_function(&hook->ops);
  if(err){
    printk("register_ftrace_function() failed: %d\n", err);
    ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    return err;
  }
  printk(KERN_INFO "PTRAC: Installed hook on %s()\n", hook->name);
  return 0;
}

void remove_hook(struct ftrace_hook *hook){
  int err;
  err = unregister_ftrace_function(&hook->ops);
  if(err)
    printk("unregister_ftrace_function() failed: %d\n", err);
  err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
  if(err)
    printk("ftrace_set_filter_ip() failed: %d\n", err);
  printk(KERN_INFO "PTRAC: Removed hook on %s()\n", hook->name);
}

// Hook definitions
static asmlinkage long (*real_sys_open)(const char __user *filename, int flags, umode_t mode);
static asmlinkage long hook_sys_open(const char __user *filename, int flags, umode_t mode){
```

```c
  long ret;
  char *kfn;

  kfn = dup_fn(filename);
  kfn = resolve_path(kfn);
  if(search_flist(kfn)) printk(KERN_INFO "PTRAC: PID %d is opening %s\n", task_pid_nr(current), kfn);
  kfree(kfn);

  ret = real_sys_open(filename, flags, mode);
  return ret;
}

static asmlinkage long (*real_sys_unlink)(const char __user *filename);
static asmlinkage long hook_sys_unlink(const char __user *filename){
  long ret;
  char *kfn;

  kfn = dup_fn(filename);
  kfn = resolve_path(kfn);
  if(search_flist(kfn)) printk(KERN_INFO "PTRAC: PID %d is unlinking %s\n", task_pid_nr(current), kfn);
  kfree(kfn);

  ret = real_sys_unlink(filename);
  return ret;
}

static asmlinkage long (*real_sys_unlinkat)(int dfd, const char __user *filename, int flag);
static asmlinkage long hook_sys_unlinkat(int dfd, const char __user *filename, int flag){
  long ret;
  char *kfn;

  kfn = dup_fn(filename);
  kfn = resolve_path(kfn);
  if(search_flist(kfn)) printk(KERN_INFO "PTRAC: PID %d is unlinking %s\n", task_pid_nr(current), kfn);
  kfree(kfn);

  ret = real_sys_unlinkat(dfd, filename, flag);
  return ret;
}

```

```
static asmlinkage long (*real_sys_rename)(const char __user *filename1, const char __user *filename2);
static asmlinkage long hook_sys_rename(const char __user *filename1, const char __user *filename2){
  long ret;
  char *kfn1, *kfn2;

  kfn1 = dup_fn(filename1);
  kfn2 = dup_fn(filename2);
  kfn1 = resolve_path(kfn1);
  kfn2 = resolve_path(kfn2);
  if(search_flist(kfn1) || search_flist(kfn2)) printk(KERN_INFO "PTRAC: PID %d is renaming %s to %s\n", task_pid_nr(current), kfn1, kfn2);
  kfree(kfn1);
  kfree(kfn2);

  ret = real_sys_rename(filename1, filename2);
  return ret;
}

static asmlinkage long (*real_sys_execve)(const char __user *filename, const char __user *const __user *argv, const char __user *const __user *envp);
static asmlinkage long hook_sys_execve(const char __user *filename, const char __user *const __user *argv, const char __user *const __user *envp){
  long ret;
  char *kfn;

  kfn = dup_fn(filename);
  kfn = resolve_path(kfn);
  if(search_flist(kfn)) printk(KERN_INFO "PTRAC: PID %d is executing %s\n", task_pid_nr(current), kfn);
  kfree(kfn);

  ret = real_sys_execve(filename, argv, envp);
  return ret;
}

// Hooks
static struct ftrace_hook open_hook = HOOK(sys_open);
static struct ftrace_hook unlink_hook = HOOK(sys_unlink);
static struct ftrace_hook unlinkat_hook = HOOK(sys_unlinkat);
static struct ftrace_hook rename_hook = HOOK(sys_rename);
static struct ftrace_hook execve_hook = HOOK(sys_execve);

// Function handlers for filelist kobject attribute
```

```c
static ssize_t filelist_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf){
  fcontrl *fcp = flist;
  int r = 0;
  int sum = 0;
  while(fcp){
    //printk(KERN_INFO "PTRAC: %d %s\n", fcp->access, fcp->fn);
    r = sprintf(buf+sum, "%d %s\n", fcp->access, fcp->fn);
    sum += r;
    fcp = fcp->next;
  }
  return sum;
}

static ssize_t filelist_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count){
  fcontrl *f;
  fcontrl *fcp = flist;
  fcontrl *prev = NULL;
  char fn[512];
  int access = 0;
  fcontrl *exists = NULL;

  printk(KERN_INFO "PTRAC: Adding to filelist: %s", buf);
  sscanf(buf, "%511s %d", fn, &access);

  // Search if exists
  while(fcp){
    if(!strcmp(fn, fcp->fn)){
      printk("File exists! ");
      exists = fcp;
      break;
    }
    prev = fcp;
    fcp = fcp->next;
  }

  if(exists){
    if(access){
      //Update
      exists->access = access;
      printk("Updated access\n");
    }else{
      //Remove element
      if(flist == exists){
        flist = exists->next;
      }else{
        prev->next = exists->next;
      }
      kfree(exists);
      printk("Removed file\n");
    }
  }else if(access){
    // Create new fcontrl
    f = (fcontrl *)kmalloc(sizeof(fcontrl), __GFP_FS);
    strncpy(f->fn, fn, 511);
    f->access = access;

    // Add to list
    f->next = flist;
    flist = f;
  }
  return count;
}

// Kobject and stuff for filelist
struct kobj_attribute kattr = __ATTR_RW(filelist);
struct kobject *kobj_ref;

static int __init ptrac_init(void){

  printk(KERN_INFO "PTRAC: Module loaded!\n");

  // Setup sysfs
  // Create new kobject and register /sys/ptrac
  kobj_ref = kobject_create_and_add("ptrac", NULL);

  // Create /sys/ptrac/filelist
  if(sysfs_create_file(kobj_ref, &kattr.attr)){
    printk(KERN_INFO "Cannot create sysfs file...\n");
    return -1;
  }

  sys_getcwd = (long (*)(char __user *buf, unsigned long size)) kallsyms_lookup_name("sys_getcwd");

  // Install hooks
  install_hook(&open_hook);
  install_hook(&unlink_hook);
  install_hook(&unlinkat_hook);
```

```c
  install_hook(&rename_hook);
  install_hook(&execve_hook);

  return 0;
}

static void __exit ptrac_exit(void){

  // Remove hooks
  remove_hook(&open_hook);
  remove_hook(&unlink_hook);
  remove_hook(&unlinkat_hook);
  remove_hook(&rename_hook);
  remove_hook(&execve_hook);

  // Decrement reference counter for /sys/ptrac
  kobject_put(kobj_ref);
  // Remove /sys/ptrac/filelist
  sysfs_remove_file(kernel_kobj, &kattr.attr);

  while(flist){
    fcontrl *fcp = flist->next;
    kfree(flist);
    flist = fcp;
  }

  printk(KERN_INFO "PTRAC: Module unloaded!\n");
}

module_init(ptrac_init);
module_exit(ptrac_exit);
```