Hooking or Monitoring System calls in linux using ftrace

nixhacker.com/hooking-syscalls-in-linux-using-ftrace

Shubham Dubey September 23, 2021

Rootkits

In this post we will see how can you use ftrace to hook linux system calls. For learning purpose, we will create a kernel module that will make any target file immutable in system.

Published 24 Sep 2021 10 min read

By Shubham Dubey

If you ever tought about hooking system calls in linux from userspace or kernel space, the most common methods you must have heard are:

1) By inject a library using `LD_PRELOAD`(user mode).

2) Modifying Syscall table(kernel mode).

But both methods have their limitations. The first mutual limitation is, both are most known/expected techniques used by attackers, so they are the first thing an AV product or researcher checks. For `LD_PRELOAD` other limitation is it's per process based (ignoring `/etc/ld.so.preload` since its very easily detectable and visible). Whereas, for syscall table modification, the biggest issue is that the table is read only since kernel version 2.6.16 (although that can be bypassed by changing `CR0.WP`, but it's an extra work and an AV detection will trigger on such sequence of code.)

A better approach which we are discussing here for syscall hooking is using ftrace. It is relatively less known and more robust method since the ftrace functionality preexist in all linux kernel above 2.6.27. So, the kernel provides full support for your operations.

Basics of Ftrace

Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel.The ftrace infrastructure was originally created to attach callbacks to the beginning of functions in order to record and trace the flow of the kernel. But these callbacks can also be used for hooking/live patching or monitoring the function calls.

Ftrace can be used to trace kernel functions call from another kernel module or can be interacted from userspace using tracefs file system and `trace-cmd` tool. To monitor system calls, we will be creating a kernel module and using ftrace functionalities inside it. To learn better how to hook a system call using ftrace, we will be creating a small fun project.

Making a file Immutable using system call hooking

Before getting into coding part, we need to understand the implementation that we have to follow to make a file non-writable/immutable. As a first thought it must come in your mind that we need to hook the `sys_write` syscall and if write operation is occurred in our target file then block it. But that method is partially complete. If you check the format of `sys_write` call:

```
long sys_write(unsigned int fd, const char __user *buf,
                          size_t count);
```

You will notice that we don't pass the filename as funtion argument, rather the call relies on file descriptor (1st arg) to identify which file to write on. You must have guessed it by now that we also need to hook `sys_open` along with `sys_write` to get the filename. In modern linux systems `sys_open` has been deprecated and replaced by `sys_openat`. So, we need to hook that rather than `sys_open`:

```
long sys_openat(int dfd, const char __user *filename, int flags,
                           umode_t mode);
```

Once we hook both calls, In `sys_openat` if we see the target filename then we need to store the *process id* and *file descriptor* of the calling process. `PID` is required since the file descriptors value can be similar between different process as the `fd` field is not global but process specific data. On every `sys_write`, we need to check if the write is done by our calling process and fd is similar or not. If it matches with our store data then we can block it. So, let's start understanding the ftrace usage and write the code in parallel.

Ftrace code can be called by importing `ftrace.h`.

```
#include <linux/ftrace.h>
```

To register a function callback, a `ftrace_ops` is required. This structure is used to tell ftrace what function should be called as the callback as well as what protections the callback will perform and not require ftrace to handle.

```
struct ftrace_ops ops = {
      .func                    = my_callback_func,
      .flags                   = MY_FTRACE_FLAGS
      .private                 = any_private_data_structure,
};
```

From this we only need to set `.func`, others are optional.

To enable and disable tracing, following calls can be used:

```
register_ftrace_function(&ops);

unregister_ftrace_function(&ops);
```

The callback function `ops->func` can be defined in the following format:

```
void callback_func(unsigned long ip, unsigned long parent_ip,
                   struct ftrace_ops *op, struct pt_regs *regs);
```

@ip This is the instruction pointer of the function that is being traced. (where the fentry or mcount is within the function)
@parent_ip This is the instruction pointer of the function that called the the function being traced (where the call of the function occurred).
@op This is a pointer to ftrace_ops that was used to register the callback. This can be used to pass data to the callback via the private pointer.
@regs If the `FTRACE_OPS_FL_SAVE_REGS` or `FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED` flags are set in the ftrace_ops structure, then this will be pointing to the pt_regs structure like it would be if an breakpoint was placed at the start of the function where ftrace was tracing. Otherwise it either contains garbage, or NULL.

If a callback is only to be called from specific functions, a filter must be set up. The filters are added by name, or ip if it is known.

```
int ftrace_set_filter(struct ftrace_ops *ops, unsigned char *buf,
                      int len, int reset);
```

Filters denote which functions should be enabled when tracing is enabled. If `buf` is NULL and reset is set, all functions will be enabled for tracing.

Using the above pieces of information, let's try to write the hook:

```
unsigned int target_fd = 0;
unsigned int target_pid = 0;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5,7,0)
static unsigned long lookup_name(const char *name)
{
        struct kprobe kp = {
                .symbol_name = name
        };
        unsigned long retval;

        if (register_kprobe(&kp) < 0) return 0;
        retval = (unsigned long) kp.addr;
        unregister_kprobe(&kp);
        return retval;
}
#else
static unsigned long lookup_name(const char *name)
{
        return kallsyms_lookup_name(name);
}
#endif

#if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
#define FTRACE_OPS_FL_RECURSION FTRACE_OPS_FL_RECURSION_SAFE
#endif

#if LINUX_VERSION_CODE < KERNEL_VERSION(5,11,0)
#define ftrace_regs pt_regs

static __always_inline struct pt_regs *ftrace_get_regs(struct ftrace_regs *fregs)
{
        return fregs;
}
#endif

/*
 * There are two ways of preventing vicious recursive loops when hooking:
 * - detect recusion using function return address (USE_FENTRY_OFFSET = 0)
 * - avoid recusion by jumping over the ftrace call (USE_FENTRY_OFFSET = 1)
 */
#define USE_FENTRY_OFFSET 0

/**
 * struct ftrace_hook - describes a single hook to install
 *
 * @name:     name of the function to hook
 *
 * @function: pointer to the function to execute instead
 *
 * @original: pointer to the location where to save a pointer
 *            to the original function
 *
 * @address:  kernel address of the function entry
 *
 * @ops:      ftrace_ops state for this function hook
 *
 * The user should fill in only &name, &hook, &orig fields.
 * Other fields are considered implementation details.
 */
struct ftrace_hook {
        const char *name;
        void *function;
        void *original;

        unsigned long address;
        struct ftrace_ops ops;
};

static int fh_resolve_hook_address(struct ftrace_hook *hook)
{
        hook->address = lookup_name(hook->name);

        if (!hook->address) {
                pr_debug("unresolved symbol: %s\n", hook->name);
                return -ENOENT;
        }

#if USE_FENTRY_OFFSET
        *((unsigned long*) hook->original) = hook->address + MCOUNT_INSN_SIZE;
#else
        *((unsigned long*) hook->original) = hook->address;
#endif

        return 0;
}

static void notrace fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
                struct ftrace_ops *ops, struct ftrace_regs *fregs)
{
        struct pt_regs *regs = ftrace_get_regs(fregs);
        struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);

#if USE_FENTRY_OFFSET
        regs->ip = (unsigned long)hook->function;
#else
        if (!within_module(parent_ip, THIS_MODULE))
                regs->ip = (unsigned long)hook->function;
#endif
}

/**
 * fh_install_hooks() - register and enable a single hook
 * @hook: a hook to install
 *
 * Returns: zero on success, negative error code otherwise.
 */
int fh_install_hook(struct ftrace_hook *hook)
{
        int err;

        err = fh_resolve_hook_address(hook);
        if (err)
                return err;

        /*
         * We're going to modify %rip register so we'll need IPMODIFY flag
         * and SAVE_REGS as its prerequisite. ftrace's anti-recursion guard
         * is useless if we change %rip so disable it with RECURSION.
         * We'll perform our own checks for trace function reentry.
         */
        hook->ops.func = fh_ftrace_thunk;
        hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                        | FTRACE_OPS_FL_RECURSION
                        | FTRACE_OPS_FL_IPMODIFY;

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

        err = register_ftrace_function(&hook->ops);
        if (err) {
                pr_debug("register_ftrace_function() failed: %d\n", err);
                ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
                return err;
        }

        return 0;
}

/**
 * fh_remove_hooks() - disable and unregister a single hook
 * @hook: a hook to remove
 */
void fh_remove_hook(struct ftrace_hook *hook)
{
        int err;

        err = unregister_ftrace_function(&hook->ops);
        if (err) {
                pr_debug("unregister_ftrace_function() failed: %d\n", err);
        }

        err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        if (err) {
                pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
        }
}

/**
 * fh_install_hooks() - register and enable multiple hooks
 * @hooks: array of hooks to install
 * @count: number of hooks to install
 *
 * If some hooks fail to install then all hooks will be removed.
 *
 * Returns: zero on success, negative error code otherwise.
 */
int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
{
        int err;
        size_t i;

        for (i = 0; i < count; i++) {
                err = fh_install_hook(&hooks[i]);
                if (err)
                        goto error;
        }

        return 0;

error:
        while (i != 0) {
                fh_remove_hook(&hooks[--i]);
        }

        return err;
}

/**
 * fh_remove_hooks() - disable and unregister multiple hooks
 * @hooks: array of hooks to remove
 * @count: number of hooks to remove
 */
void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
{
        size_t i;

        for (i = 0; i < count; i++)
                fh_remove_hook(&hooks[i]);
}



#define HOOK(_name, _function, _original)       \
        {                                       \
                .name = SYSCALL_NAME(_name),    \
                .function = (_function),        \
                .original = (_original),        \
        }

static struct ftrace_hook demo_hooks[] = {
        HOOK("sys_write", fh_sys_write, &real_sys_write),
        HOOK("sys_openat", fh_sys_openat, &real_sys_openat),
};

static int fh_init(void)
{
        int err;

        err = fh_install_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));
        if (err)
                return err;

        pr_info("module loaded\n");

        return 0;
}
module_init(fh_init);

static void fh_exit(void)
{
        fh_remove_hooks(demo_hooks, ARRAY_SIZE(demo_hooks));

        pr_info("module unloaded\n");
}
module_exit(fh_exit);
```

`demo_hooks` function have two syscalls listed `SYS_OPENAT` and `SYS_WRITE` as we decided above.

We have a `lookup_name` function that will return the address of system call in kernel memory.

Let's check how our callback will looks like. From callback we can call the original function and return the value received from original.

```
/*
 * x86_64 kernels have a special naming convention for syscall entry points in newer
kernels.
 * That's what you end up with if an architecture has 3 (three) ABIs for system
calls.
 */
#ifdef PTREGS_SYSCALL_STUBS
#define SYSCALL_NAME(name) ("__x64_" name)
#else
#define SYSCALL_NAME(name) (name)
#endif

#ifdef PTREGS_SYSCALL_STUBS
static asmlinkage long (*real_sys_write)(struct pt_regs *regs);

static asmlinkage long fh_sys_write(struct pt_regs *regs)
{
        long ret;

        ret = real_sys_write(regs);

        return ret;
}
#else
static asmlinkage long (*real_sys_write)(unsigned int fd, const char __user *buf,
                 size_t count);

static asmlinkage long fh_sys_write(unsigned int fd, const char __user *buf,
                 size_t count)
{
        long ret;

        ret = real_sys_write(fd, buf, count);

        return ret;
}
#endif


#ifdef PTREGS_SYSCALL_STUBS
static asmlinkage long (*real_sys_openat)(struct pt_regs *regs);

static asmlinkage long fh_sys_openat(struct pt_regs *regs)
{
        long ret;

        ret = real_sys_openat(regs);

        return ret;
}
#else
static asmlinkage long (*real_sys_openat)(int dfd, const char __user *filename,
                                int flags, umode_t mode);

static asmlinkage long fh_sys_openat(int dfd, const char __user *filename,
                                int flags, umode_t mode)
{
        long ret;
        ret = real_sys_openat(filename, flags, mode);
        return ret;
}
#endif
```

Our logic for making file non-writable will comes under `fh_sys_openat` and `fh_sys_write`.

Finding the calling process's pid: We need to save two things in `fh_sys_openat`. First is `fd` which will be returned from `read_sys_openat`. Next is `pid`, which you will get from `current` task structure.

```
struct task_struct *task;
task = current;
int pid = task->pid
```

using this, let's write the logic for `fh_sys_openat`

```
unsigned int target_fd = 0;
unsigned int target_pid = 0;


static char *duplicate_filename(const char __user *filename)
{
        char *kernel_filename;

        kernel_filename = kmalloc(4096, GFP_KERNEL);
        if (!kernel_filename)
                return NULL;

        if (strncpy_from_user(kernel_filename, filename, 4096) < 0) {
                kfree(kernel_filename);
                return NULL;
        }

        return kernel_filename;
}

#ifdef PTREGS_SYSCALL_STUBS
static asmlinkage long (*real_sys_openat)(struct pt_regs *regs);

static asmlinkage long fh_sys_openat(struct pt_regs *regs)
{
        long ret;
        char *kernel_filename;
        struct task_struct *task;
        task = current;

        kernel_filename = duplicate_filename((void*) regs->si);
        if (strncmp(kernel_filename, "/tmp/test.txt", 13) == 0)
        {
                pr_info("our file is opened by process with id: %d\n", task->pid);
                pr_info("opened file : %s\n", kernel_filename);
                kfree(kernel_filename);
                ret = real_sys_openat(regs);
                pr_info("fd returned is %ld\n", ret);
                target_fd = ret;
                target_pid = task->pid;
                return ret;

        }

        kfree(kernel_filename);
        ret = real_sys_openat(regs);

        return ret;
}
#else
```

```
static asmlinkage long fh_sys_openat(int dfd, const char __user *filename,
                                int flags, umode_t mode)
{
        long ret;
        char *kernel_filename;
        struct task_struct *task;
        task = current;

        kernel_filename = duplicate_filename(filename);
        if (strncmp(kernel_filename, "/tmp/test.txt", 13) == 0)
        {
                pr_info("our file is opened by process with id: %d\n", task->pid);
                pr_info("opened file : %s\n", kernel_filename);
                kfree(kernel_filename);
                ret = real_sys_openat(dfd, filename, flags, mode);
                pr_info("fd returned is %ld\n", ret);
                target_fd = ret;
                target_pid = task->pid;
                return ret;

        }

        kfree(kernel_filename);

        ret = real_sys_openat(filename, flags, mode);

        return ret;
}
#endif
```

We are saving the `pid` and `fd` recieved from openat call into `target_pid` and `target_fd` global variable.

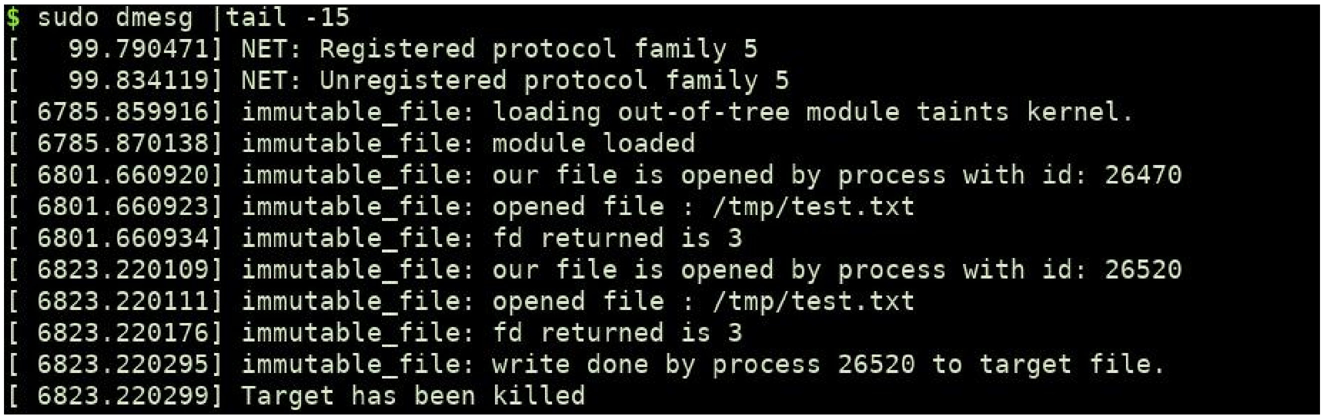
As an extra step, we will kill the calling process when they try to write to our monitored file. To kill a process from kernel module, we need to send `SIGKILL` to the process. The code for that will looks like this:

```
struct task_struct *task;
task = current;
int signum = SIGKILL;
struct kernel_siginfo info;

memset(&info, 0, sizeof(struct kernel_siginfo));
info.si_signo = signum;
int ret = send_sig_info(signum, &info, task);
```

Our complete code for `fh_sys_write` will look like this:

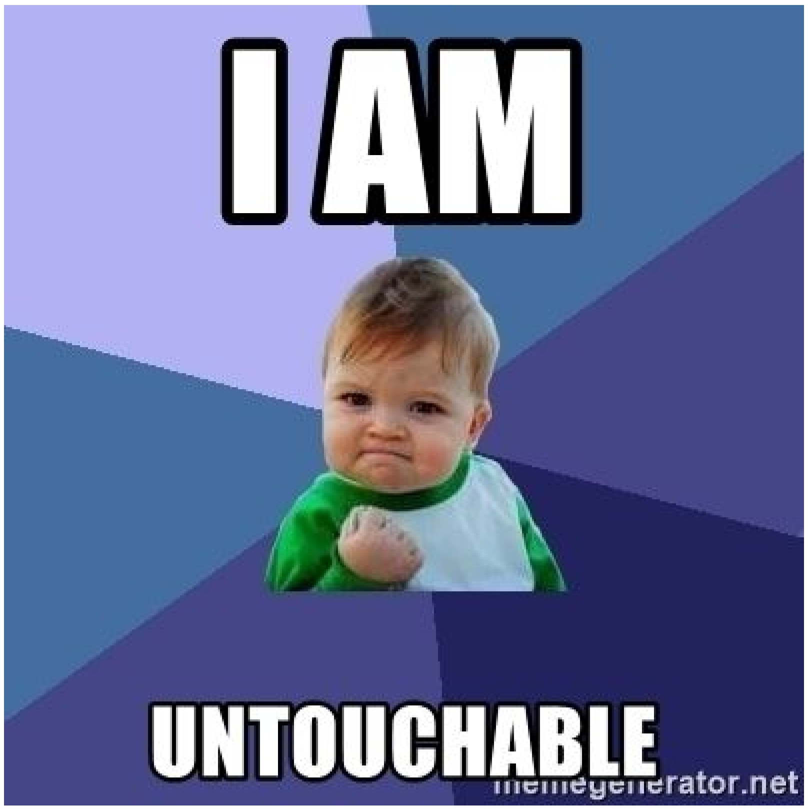
```
#ifdef PTREGS_SYSCALL_STUBS
static asmlinkage long (*real_sys_write)(struct pt_regs *regs);

static asmlinkage long fh_sys_write(struct pt_regs *regs)
{
        long ret;
        struct task_struct *task;
        task = current;
        int signum = SIGKILL;
        if (task->pid == target_pid)
        {
                if (regs->di == target_fd)
                {
                        pr_info("write done by process %d to target file.\n", task-
>pid);
                        struct kernel_siginfo info;
                        memset(&info, 0, sizeof(struct kernel_siginfo));
                        info.si_signo = signum;
                        int ret = send_sig_info(signum, &info, task);
                                        if (ret < 0)
                                        {
                                          printk(KERN_INFO "error sending signal\n");
                                        }
                                        else
                                        {
                                                printk(KERN_INFO "Target has been
killed\n");
                                                return 0;
                                        }
                }
        }

        ret = real_sys_write(regs);

        return ret;
}
#else
static asmlinkage long (*real_sys_write)(unsigned int fd, const char __user *buf,
                 size_t count);

static asmlinkage long fh_sys_write(unsigned int fd, const char __user *buf,
                 size_t count)
{
        long ret;
        struct task_struct *task;
        task = current;
        int signum = SIGKILL;
        if (task->pid == target_pid)
        {
                if (fd == target_fd)
                {
                        pr_info("write done by process %d to target file.\n", task-
>pid);

                        struct kernel_siginfo info;
                        memset(&info, 0, sizeof(struct kernel_siginfo));
                        info.si_signo = signum;
                        int ret = send_sig_info(signum, &info, task);
                                        if (ret < 0)
                                        {
                                          printk(KERN_INFO "error sending signal\n");
                                        }
                                        else
                                        {
                                                printk(KERN_INFO "Target has been
killed\n");
                                                return 0;
                                        }
                }
        }

        ret = real_sys_write(fd, buf, count);


        return ret;
}
#endif
```

This is all you need to do to stop any process from modifying a file. You can find the complete code here. Once you clone the repo, run `make` command and try to write to `/tmp/test.txt` (default location). You will see something like this in `dmesg` logs.



```
$ sudo dmesg |tail -15
[   99.790471] NET: Registered protocol family 5
[   99.834119] NET: Unregistered protocol family 5
[ 6785.859916] immutable_file: loading out-of-tree module taints kernel.
[ 6785.870138] immutable_file: module loaded
[ 6801.660920] immutable_file: our file is opened by process with id: 26470
[ 6801.660923] immutable_file: opened file : /tmp/test.txt
[ 6801.660934] immutable_file: fd returned is 3
[ 6823.220109] immutable_file: our file is opened by process with id: 26520
[ 6823.220111] immutable_file: opened file : /tmp/test.txt
[ 6823.220176] immutable_file: fd returned is 3
[ 6823.220295] immutable_file: write done by process 26520 to target file.
[ 6823.220299] Target has been killed
```

Now, every time someone tried to modify the file, the modification will fail and the process will get killed.



References:

https://www.kernel.org/doc/html/v4.17/trace/ftrace-uses.html

Hooking Linux Kernel Functions, Part 2: How to Hook Functions with Ftrace

Learn how to use ftrace - a popular Linux feature - for hooking critical kernel function calls.

AprioritSuper User

Github Repo link:
https://github.com/shubham0d/Immutable-file-linux