Distribution Kernel Security Hardening with ftrace

Because sometimes your OS vendor just doesn't have the security features that you want.

Written by:
Corey Henderson

Exploit Attack Surface

Hardening system security is essentially all about reducing the attack surface in which a bad actor is able work in. In the absence of hardening tools, the default mode of the linux operating system is discretionary access control, which allows programs to make any syscall available on the system. While syscalls are generally well protected, little is done to detect and eliminate arbitrary code execution that occurs due to overran memory, which is largely where privilage escalation exploits are made possible.

Exploits that give elevated access on a system are often done through syscalls in which a program shouldn't need to make anyway. Even in the presence of mandatory access control systems such as SELinux and AppArmor, access to a wide array of syscalls are still left unrestricted.

The Linux Security Module (LSM) framework that allows these mandatory access control systems still doesn't directly address this either. While tightening what access a process has to the filesystem and other running processes, it doesn't allow for targeted instrumented protections against available syscalls. Thus, if a bad actor is able to execute arbitrary code in memory, the LSM framework has little ability to detect and stop the attempt.

The grsecurity project does an excellent job in preventing exploits via address randomization, preventing memory overrun, execution of read-only memory, among other methods, yet its protections are mostly unavailable in most linux kernel distributions available today.

Firewall for syscalls

The seccomp filter interface does allow a process to drop its ability to make syscalls, akin to how a firewall prevents the network from talking to specific ports. Docker has done an excellent job of making this easily available to its container process, and will go a long way to narrowing the attack surface of any program it isolates. However, this does nothing if any exploit is available in a syscall that a program does have access to, and the default docker seccomp configuration is still rather permissive.

Consider running an application in a Docker container process with all kernel capabilities dropped, a read-only filesystem, and a seccomp profile that limits the available syscalls down to a narrow set. This is a very small attack surface should the container process be compromised. However, a bad actor still has the non-filtered syscalls avaiable for exploit. Like any firewall, an available syscall is akin to a port allowed by a firewall.

Advanced firewalls are able to monitor traffic on allowed ports and take defensive action if that traffic is suspicious. The generic seccomp filter doesn't have this capability, and the LSM framework is limited in its scope. While this is a least privileged scenario, there are still opportunities for a bad actor to do privilege escalation, because some syscalls are still available for use. How do we instrument advanced functionality for active checking of what a process does send through these syscalls?

Hooking kernel functions with ftrace / fopskit

The ftrace interface is intended for tracing kernel threads for debugging purposes, but can do so much more than that. Because it can instrument almost any kernel symbol to redirect the execution flow of any process on the system, it can be used to hook kernel functions that otherwise have insufficient capability for targeted projection. The out-of-kernel “fopskit” code is a wrapper to ftrace to make the hooking of kernel symbols easy to do from within a kernel module.

Below is code for a very simple example of this. It targets the `sys_open()` syscall with code to deny an open if the running process is in a given group id and the file is within the /boot directory. While this can be done with normal filesystem permissions, it shows how targeted code written by a security programmer to instrument specific protections to specific kernel functions.

First, download the source code of “fopskit” located in on github:

https://github.com/cormander/tpe-lkm/blob/2.0.0/fopskit.h
https://github.com/cormander/tpe-lkm/blob/2.0.0/fopskit.c

Then compile the below module code:

```
#include <linux/module.h>
#include "fopskit.h"

int test_eaccess(void) {
        return -EACCES;
}

fopskit_hook_handler(sys_open) {
        if (in_group_p(KGIDT_INIT(1000)) &&
                !strncmp((const char *)REGS_ARG1, "/boot", 5))
                regs->ip = (unsigned long)test_eaccess;
}

struct fops_hook hook_sys_open = fops_hook_val(sys_open);

static int __init test_init(void) {
        fopskit_sym_hook(&hook_sys_open);
        return 0;
}

static void __exit test_exit(void) {
        fopskit_sym_unhook(&hook_sys_open);
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

After module insertion, a user belonging to gid 1000 can no longer make open calls on files in the /boot directory:

```
$ sudo insmod test.ko
$ cat /boot/config-3.10.0-514.10.2.el7.x86_64
cat: /boot/config-3.10.0-514.10.2.el7.x86_64
$ echo $?
1
```

This of course isn't a complete protection of the files in the boot directory, as other syscalls can still find their way into reading those files, but serves as an example of how easy it is and how little code is required to hook a function with ftrace.

Instrumenting Security Hardening Features

Consider the "Trusted Path Execution" (TPE) feature of grsecurity. It's a simple concept; users are denied execution of binary code that exist in files they are able to write to. This prevents certain entry points for code by a bad actor that would otherwise be left wide open for exploitation.

Since almost every process on a system needs the ability to make calls to the syscalls responsible for code execution (mmap, mprotect, and execve), removing access to them via seccomp filtering is unrealistic. The LSM hooks also fail to protect in this area because they are limited to the code that already exists in the kernel itself, which does not add the ability to create this protection.

By planting an ftrace at functions used by each of these syscalls and redirecting to new code by a loadable kernel module, a system administrator can instrument the targeted protection of TPE with very minimal code. Consider the following example:

```
fopskit_hook_handler(security_bprm_check) {
        struct linux_binprm *bprm = (struct linux_binprm *)REGS_ARG1;

        if (bprm->file)
                if (tpe_allow_file(bprm->file, "exec"))
                        regs->ip = (unsigned long)tpe_eaccess;
}
```

What happens here is the `fopskit_hook_handler()` macro automatically creates the needed `fops_struct` corresponding to the kernel function, in this case, `security_bprm_check()`, and assigns its address to it. The `fopskit_sym_hook()` call uses the ftrace interface to redirect the given kernel symbol back to that code.

The hook handler is a void function and normally ftrace would send the code flow back to the original function, preventing the change of the return value of the function. However, you can override the kernel return address via `regs->ip` register, and thus have the ability to control the return value. In this example, if the `tpe_allow_file()` check return an error, the code flow is redirected to the `tpe_eaccess()` function, which returns the "permission denied" error code back to the calling thread.

This only handles calls to execve. Full protection of code execution of course requires instrumentation of the mmap and mprotect syscalls, which the “tpe-lkm” kernel module project does. Full source code available on github:

https://github.com/cormander/tpe-lkm/

Summary

By closing down the unused syscalls of a process via the seccomp filter, and hooking the syscalls left open, security programmers can instrument targeted code to create protections akin to an advanced firewall. While the container process needs certain syscalls to function, the ability to lock down those syscalls to only the intended purposes can further secure a system from bad actors. Also, security features that are out-of-tree of that distribution could be added. For example, AppArmor could theoretically be inserted into a kernel running SELinux, where under normal circumstances only one LSM framework could be run.