**Hooking Functions**

Hooking is when you use a handler function (the *hook*) to modify the control flow of a program. A hook will register its address as the location for a specific function, so that when that function is called, the hook is run instead. In our context, we use this process to change the results of an OS’s API.

Importantly, while a knowledge of system calls and how they work is critical here, you aren’t actually writing your own system call modules from scratch. This would mean creating something a user could run freely, registering it in the system calls table, etc. What we want to do here is just create a regular KLD, load that module into kernel with DECLARE\_MODULE, and then have it so that that module is what an *already existing* system call points to (instead of its original target module.) The SYSCALL\_MODULE macro is responsible for setting up all of the stuff with the system table, but that’s already done for us – we’re just overriding one part of it.

So, it turns out that with a somewhat strong understanding of system calls, function hooking is actually substantially easier than I anticipated. Let’s dive into it.

**Hooking a System Call**

So, all applications which need to access the kernel in some way will make use of a system call. In FreeBSD, this is facilitated by a table of sysent structures, which essentially gives the system an overview of all linked/registered system call functions.

It follows that in order to change the control flow of one of these calls, all we need to do is change which function a particular *sysent[]* entry points to. Want to modify *$mkdir*? When your rootkit module loads, go to mkdir’s entry in the system call table, find its *sy\_call* pointer and change its value to instead point at our own hook function. If you want, you can do the inverse on unload.

And voila, you have a basic system call hook.

**Tracing a Kernel Process**

The obvious question remains: if we now know how to hook into any system call, how do we actually figure out which entries to *sysent[]* we need to modify to hook into/modify a specific process? The answer to this is *Kernel Process Tracing*.

This is a diagnostic technique, which intercepts each kernel operation performed on behalf of a specific running process. This is done with *ktrace* (1) and *kdump* (1). Trace enables tracing for a specific process, while dump will then display the trace data.

When you know the name of the system call, you can find it in sysent[] with the following macro: SYS\_[syscall\_name]. E.G: read 🡪 sysent[SYS\_read]; mkdir 🡪 sysent[SYS\_mkdir].

**Common SysCall Hooks**

Here are some common system call hooks that I grabbed from *Designing BSD Rootkits*:

| **System Call** | **Purpose of Hook** |
| --- | --- |
| read, readv, pread, preadv | Logging input |
| write,writev,pwrite, pwritev | Logging output |
| open | Hiding file contents |
| unlink | Preventing file removal |
| chdir | Preventing directory traversal |
| chmod | Preventing file mode modification |
| chown | Preventing ownership change |
| kill | Preventing signal sending |
| ioctl | Manipulating ioctl requests |
| execve | Redirecting file execution |
| rename | Preventing file renaming |
| rmdir | Preventing directory removal |
| stat, lstat | Hiding file status |
| getdirentries | Hiding files |
| truncate | Preventing file truncating or extending |
| kldload | Preventing module loading |
| kldunload | Preventing module unloading |