

Lab 4



- ✦ You will write a Linux kernel module
- ✦ Linux is written in C, but does not include all standard libraries
 - ✦ And some other idiosyncrasies
- ✦ This lecture will give you a crash course in writing Linux kernel code

Kernel Modules



- ✦ Sort of like a dynamically linked library
- ✦ How different?
 - ✦ Not linked at load (boot) time
 - ✦ Loaded dynamically
 - ✦ Often in response to realizing a particular piece of hardware is present on the system
 - ✦ For more, check out `udev` and `lspci`
 - ✦ Built with `.ko` extension (kernel object), but still an ELF binary

Kernel Modules, cont.



- ✦ Load a module
 - ✦ insmod – Just load it
 - ✦ modprobe – Do some dependency checks
 - ✦ Examples?
 - ✦ rmmod – Remove a module
- ✦ Module internally has init and exit routines, which can in turn create device files or otherwise register other call back functions

Events and hooks



- ✦ When you write module code, there isn't a `main()` routine, just `init()`
- ✦ Most kernel code is servicing events---either from an application or hardware
- ✦ Thus, most modules will either create a device file, register a file system type, network protocol, or other event that will lead to further callbacks to its functions

Kernel Modules, cont.



- ✦ When a module is loaded, it runs in the kernel's address space
 - ✦ And in ring 0
- ✦ So what does this say about trust in this code?
 - ✦ It is completely trusted as part of the kernel
- ✦ And if this code has a bug?
 - ✦ It can crash the kernel

Accessing Kernel Functions

- ✦ Linux defines public and private functions (similar to Java)
 - ✦ Look for “EXPORT_SYMBOL” in the Linux source
- ✦ Kernel exports a “jump table” with the addresses of public functions
 - ✦ At load time, module’s jump table is connected with kernel jump table
- ✦ But what prevents a module from using a “private” function?
 - ✦ Nothing, except it is a bit more work to find the right address
 - ✦ Example code to do this in the lab4 handout

Kernel Programming



- ✦ Big difference: No standard C library!
 - ✦ Sound familiar from lab 1?
 - ✦ Why no libc?
- ✦ But some libc-like interfaces
 - ✦ malloc -> kmalloc
 - ✦ printf("boo") -> printk(KERN_ERR "boo")
- ✦ Some things are missing, like floating point division

Kernel Programming, ctd



- ✦ Stack can't grow dynamically
 - ✦ Generally limited to 4 or 8KB
 - ✦ So avoid deep recursion, stack allocating substantial buffers, etc.
- ✦ Why not?
 - ✦ Mostly for simplicity, and to keep per-thread memory overheads down
 - ✦ Also, the current task struct can be found by rounding down the stack pointer (esp/rsp)

Validating inputs super-important!

- ✦ Input parsing bugs can crash or compromise entire OS!
- ✦ Example: Pass `read()` system call a null pointer for buffer
 - ✦ OS needs to validate that buffer is really mapped
- ✦ Tools: `copy_from_user()`, `copy_to_user()`, `access_ok()`, etc.

Cleaning up



- ✦ After an error, you have to be careful to put things back the way you found them (generally in reverse order)
 - ✦ Release locks, free memory, decrement ref counts, etc.
- ✦ The `_one_` acceptable use of `goto` is to compensate for the lack of exceptions in C

Clean Up Example



```
str = getname(name);
if (IS_ERR(str)) {
    err = -EFAULT;
    printk (KERN_DEBUG "hash_name: getname(str) error!\n");
    goto out;
}

if (!access_ok(VERIFY_WRITE, hash, HASH_BYTES)) {
    err = -EFAULT;
    printk (KERN_DEBUG "hash_name: access_ok(hash) error!\n");
    goto putname_out;
}

// helper function does all the work here
putname_out:
    putname(str);
out:
    return err;
}
```

Key objects

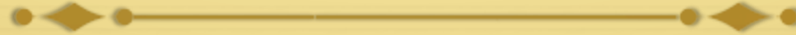


- ✦ `task_struct` – a kernel-schedulable thread
 - ✦ `current` points to the current task
- ✦ `inode` and `dentry` – refer to a file's `inode` and `dentry`, as discussed in the VFS lectures
 - ✦ Handy to find these by calling helper functions in the `fs` directory
 - ✦ Read through `open` and friends

Object-orientation in the VFS

- ✦ Files have a standard set of operations
 - ✦ Read, write, truncate, etc.
- ✦ Each inode includes a pointer to a 'file_operations' struct
 - ✦ Which in turn points to a lot of functions
- ✦ VFS code is full of things like this:
 - ✦ `int rv = inode->f_op->stat(inode, statbuf);`

OO, cont.



- ✦ When an inode is created for a given file system, the file system initializes the `file_operation` structure
- ✦ For lab 4, you may find it handy to modify/replace a given file's `file_operation` structure

/proc



- ✦ The kernel exports a lot of statistics, configuration data, etc. via this pseudo-file system
- ✦ These “files” are not stored anywhere on any disk
- ✦ The kernel just creates a bunch of inodes/dentries
 - ✦ And provides read/write and other file_operations hooks that are backed by kernel-internal functions
 - ✦ Check out fs/proc source code

Logs?



- ✦ The kernel log goes into `/var/log/dmesg` by default
 - ✦ And to the console
 - ✦ Visible in vsphere for your VM
- ✦ Also dumped by the `dmesg` command
- ✦ `printk` is your friend for debugging!

Verbosity



- ✦ The kernel is dynamically configured with a given level of verbosity in the logs
- ✦ The first argument to `printk` is the importance level
 - ✦ `printk(KERN_ERR "I am serious");`
 - ✦ `printk(KERN_INFO "I can be filtered");`
- ✦ This style creates an integer that is placed at the front of the character array, and transparently filtered
- ✦ For your debugging, just use a high importance level

Lists



- ✦ Linux embeds lists and other data structures in the objects, rather than dynamically allocate list nodes
- ✦ Check out `include/linux/list.h`
- ✦ It has nice-looking macro loops like `list_for_each_entry`
- ✦ In each iteration, it actually uses compiler macros to figure out the offset from a next pointer to the “top” of a struct

Assertions



- ✦ `BUG_ON(condition)`
- ✦ Use this.
- ✦ How does it work?
 - ✦ if `(condition)` crash the kernel;
 - ✦ It actually uses the 'ud2a' instruction, which is a purposefully undefined x86 instruction that will cause a trap
 - ✦ The trap handler can unpack a more detailed crash report

Other tips



- ✦ Snapshot your VM for quick recreation if the file system is corrupted
- ✦ Always save your code on another machine before testing
 - ✦ `git push` is helpful for this
- ✦ Write defensively: lots of test cases and assertions, test each line you write carefully
 - ✦ Anything you guess might be true, add an assertion

Good luck!

