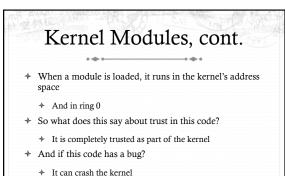


- * When you write module code, there isn't a main() routine, just init()
- + Most kernel code is servicing events---either from an application or hardware
- * Thus, most modules will either create a device file, register a file system type, network protocol, or other event that will lead to further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



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_form_user(), copy_to_user(), access_ok(), etc.

