## Loading User-Written Modules into the Kernel

**NB: This module file is laid out** [**here**](https://www.freebsd.org/cgi/man.cgi?query=module&sektion=9)**.**

Our current way of loading user-written code into the kernel is through a loadable kernel module. This is a feature which allows sysadmins to dynamically modify the kernel code, so will be great (at least at this stage) for loading our rootkit code to test.

Whenever one of these is loaded/unloaded, the kernel will run a function called the “module event handler,” which handles the start-up and shutdown routines for the module. Because of this, every loadable module must contain an event handler.

This event handler function has the following prototype:

typedef int (\*modeventhand\_t)(module\_t, int /\* modeventtype\_t \*/, void \*)&semi&semi&semi&semi;

i.e. We typedef modeventhand\_t  to represent a function with the following parameters and the return type int.

To explain each parameter:

* module\_t points to a module struct. For our hello\_world program, where there is no actual module we are loading in, we can simply pass in a blank struct.
* modeventtype\_t is passed in an enum by the system, which can be one of 4 things: MOD\_LOAD, MOD\_UNLOAD, MOD\_SHUTDOWN, MOD\_QUIESCE. It basically represents what the system is actually doing with the module you’re loading in.
* The pointer is just any additional arguments the loader gives to the function.

I.E – for a working kernel module, we must have a function resembling this prototype in our file.

**The question now is, how do we load this into the kernel?** That is to say, which part of our code is actually saying: ‘Hey, this function here is what I want you to do during the loading/unloading events.’ That’s where the DECLARE\_MODULE macro comes in. Its prototype looks like this:

#define DECLARE\_MODULE(name, data, sub, order)

Writing [DECLARE\_MODULE](https://www.freebsd.org/cgi/man.cgi?query=DECLARE_MODULE&sektion=9&n=1) (…) at the bottom of our file will allow us to:

1. Declare a generic kernel module&semi&semi&semi&semi&semi&semi; and
2. Register the module with the system (through the SYSINIT macro, which I think is called within this macro)

The parameters mean the following:

* Name: The generic module name. This doesn’t need to match anything in your code file. It simply is used in the following SYSINIT() call to identify the module in plaintext.
* Data: This is another struct: moduledata\_t. This contains: official name (of the module) which will be used in the module\_t struct & a pointer to the event handler function. (I.E. The function we wrote to handle event loading.)
  + This looks like this:



* The last two parameters aren’t as important – they just deal with the module type and when it’s loaded into the kernel, respectively.

Putting all of these together, you have enough to write a basic kernel module! Now, you just need to compile this with a basic makefile into a .ko and load this as root user with KLDLOAD().