COMP6441

SOMETHING AWESOME

*ROOTKITS*

Roark Menezes

z5116870

Contents

**My Proposal………………………………………………………………………………………………………………………… 3**

**Introduction………………………………………………………………………………………………………………………….5**

**My Goals and Performance…………………………………………………………………………………………………..5**

**Core Goals……………………………………………………………………………………………………………………………6**

**Chapter 1…………………………………………………………………………………………………………………6**

**Chapter 2………………………………………………………………………………………………………………..14**

**Chapter 3………………………………………………………………………………………………………………..21**

**Chapter 4………………………………………………………………………………………………………………..33**

**Chapter 5………………………………………………………………………………………………………………..40**

**Chapter 6………………………………………………………………………………………………………………..53**

**Extra Technical Research/Work…………………………………………………………………………………………..61**

**FreeBSD and Samba Installation……………………………………………………………………………..61**

**Brief Introduction to OS Theory………………………………………………………………………………64**

**Trojan Vs Rootkit…………………………………………………………………………………………………….65**

**My First System Call………………………………………………………………………………………………..66**

**How to Allocate Kernel Space………………………………………………………………………………….68**

**How to Test a Character Device……………………………………………………………………………….70**

**Prevent Kernel Data Corruption……………………………………………………………………………….72**

**Hiding a Running Process…………………………………………………………………………………………72**

**Semi Working Rootkit……………………………………………………………………………………………..78**

**Conclusion………….……………………………………………………………………………………………………………….81**

My Proposal

For my Something Awesome project I will be understanding the process, functionality and uses of a rootkit and then proceed to developing my own rootkit. Since I am a COMP6441 student, I dont expect to reach the D or HD criteria, but I will definitely know how far I can go once I read the book. There are many examples in the book that will aid my understanding of certain concepts. Through the weekly blog posts I will demonstrate this understanding. As I research these concepts, I will understand the difficulty level of some of the tasks below and, as a result, continually edit this proposal to make my goals clearer for the marker and for myself.

**Initial Goals**

1. Read "Developing BSD Rootkits - An Introduction to Kernel Hacking" by Joseph Kong.
2. Generate an understanding of what a rootkit is and solidify this understanding by creating a short explanation including the process of creating one.
3. As the book explains how to code rootkits in BSD, compare this with the rootkits found on other OS's (e.g. Linux, Windows etc.)
4. Generate blog posts when problems are encountered understanding the book's concepts and give detailed explanations on how they were solved.
5. Give 1 in-depth example every week of when a rootkit was used in the real world.
6. Create the rootkit, along with at least 1 blog post weekly summarising the work that was done.
7. Create at least 1 additional blog post every week detailing a problem that was encountered and how the solution was obtained.
8. Demonstrate the rootkits functionality on a personal computer.

**Extended Goals**

1. Create/Import a piggybacking trojan for the rootkit and demonstrate its functionality on a personal computer.
2. Install the rootkit on the personal computer without the use of a physical device (e.g. USB)

**Method**

WEEK 3-4

1. Read "Developing BSD Rootkits - An Introduction to Kernel Hacking" by Joseph Kong.
2. Make weekly blog posts detailing what chapters were read and what concepts were understood.

WEEK 5-8

1. Create the rootkit, making blog posts every time a milestone was passed (milestones will be created once the book is read and concepts are fully understood).
2. Create the piggybacking trojan (or import it depending on the time left; either way, make blog posts analysing the functionality of the trojan)
3. Find a way to install the rootkit on a personal computer without a physical device.
4. Test functionality of the rootkit (and trojan if complete) from a remote location.

WEEK 9

1. Create a presentation summarizing the entire project (~10 minutes).

**Criteria**

* **PS**
  + **​**Book read in its entirety, weekly blog posts, and a weekly real-world rootkit example described in depth.
* **CR**
  + **​**An attempt was made at the rootkit, full functionality however is not present.
  + An attempt was made at milestone blog posts.
  + Some research was done on creating the trojan.
* **D**
  + **​**Rootkit functionality is present, root access of the remote computer can be demonstrated (at least 1 of the following)
    - File transfer/removal
    - Application installation
    - File editing
  + Research was done on the trojan horse and an attempt was made at creating it **OR**the trojan was imported
    - Some blog posts present analysing the functionality of the trojan
* **HD**
  + **​**
  + **​**Rootkit functionality is present, root access of the remote computer can be demonstrated (all of the following)
    - File transfer/removal
    - Application installation
    - File editing
  + Trojan horse completely working
    - Functionality can be demonstrated after installing the rootkit
    - Trojan horse process appears hidden on remote computer

**Introduction**

As part of my "Something Awesome" project, I decided to research the concept of the rootkit. It was something that I've wanted to do for a long time, but never had the technical adeptness for. But after 3 years of Computer Science study, I felt I was ready to dive into the world of kernels and rootkits. Even though I am part of the 6441 stream, I wanted to take this project to test my limits.

My Goals and Performance

As per my initial goals, I decided that the best way to go about researching a rootkit was to also develop one, though not a completely functioning one. I felt that this was okay as long as I understood the concepts behind the functionality of the rootkit. This is exactly what the book provided me with. After studying and understanding every concept, I applied it, by coding programs in the FreeBSD OS (evidence provided in technical section). The semi-functioning rootkit is the combination of these little programs that I developed. Every single concept was new to me, however my previous study of COMP2121 (Assembly Code) and COMP3331 (Socket Programming) did help a lot in understanding the concepts of x86 call statements and hiding an open TCP-based port.

After reading the book, and understanding all the concepts presented, I went into researching a real-world example of where a rootkit was used. I decided on Stuxnet, as it was previously mentioned in the lectures and got my attention straight away as it involved the Israeli and U.S. governments: very exciting stuff. Along the way, I also engaged in activities outside the scope of my proposal. This included tasks such as setting up Samba (the FreeBSD file sharing service), researching on OS theory and many other kernel manipulation tasks. Below are links to the work that I have done as part of this project.

Core Goals

# Book Concept Summaries

## Chapter 1 – Loadable Kernel Modules

### KLD’s

The first exercise presented in the book was to load a piece of code into the running kernel of the system. This involved making use of a module event handler and the DECLARE\_MODULE macro.

An event handler is called whenever a KLD is loaded into/unloaded from the kernel. It handles the initialisation and shutdown routines for the KLD. ***Every single KLD must include an event handler (unless the KLD is just a sysctl).*** It is defined in the sys/module.h header and has the following structure:

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

module\_t is a pointer to a module structure and modeventtype\_t is defined as (in the same header):

*typedef enum modeventtype {*  
*MOD\_LOAD, /\* Set when module is loaded. \*/*  
*MOD\_UNLOAD, /\* Set when module is unloaded. \*/*  
*MOD\_SHUTDOWN, /\* Set on shutdown. \*/*  
*MOD\_QUIESCE /\* Set on quiesce. \*/****quiesce means to pause***  
*} modeventtype\_t;*

This is an enum that can be checked for whether is module is being loaded or unloaded. We can make an example load function that prints to the console when it has been loaded and unloaded as such *(semi colons omitted as they gave errors in open learning)*:

static int load(struct module \*module, int cmd, void \*arg){

int error = 0

switch(cmd){

case MOD\_LOAD: //when the module has loaded

uprintf("Loaded.")

break

case MOD\_UNLOAD: //when the module has deloaded

uprintf("Unloaded.");

break

default: //either shutdown or quiesce

error = EOPNOTSUPP (Error Operation Not Supported)

break

}

return(error)

}

This defines our KLD, however the next part of the puzzle is to figure out how to ling and register this KLD with the kernel. To do this we must call the DECLARE\_MODULE macro which is defined in the same module.h header as such

***#define DECLARE\_MODULE(name, data, sub, order)*** *MODULE\_METADATA(\_md\_##name, MDT\_MODULE, &data, #name);   
SYSINIT(name##module, sub, order, module\_register\_init, &data)   
struct  \_\_hack*

Here is a brief description of the 4 parameters:

* name - the generic module name, passed as a string
* data - the official module name and event handler function, passed as a *moduledata*structure.
  + typedef struct moduledata {  
    *const char* \*name; /\* module name \*/  
    *modeventhand\_t evhand*; /\* event handler \*/  
    *void \*priv*; /\* extra data \*/  
    } moduledata\_t;
* sub - system start-up interface specifier, entries **found in the sysinit\_sub\_id enum in sys/kernel.h**. *For this project we use SI\_SUB\_DRIVERS, which is used when registering a device driver*
* order - specifies the order of initialisation within the subsystem, entries **found in sysinit\_elem\_order in sys/kernel.h.***For this project we will use SI\_ORDER\_MIDDLE, which will initialise the KLD somewhere in the middle.*

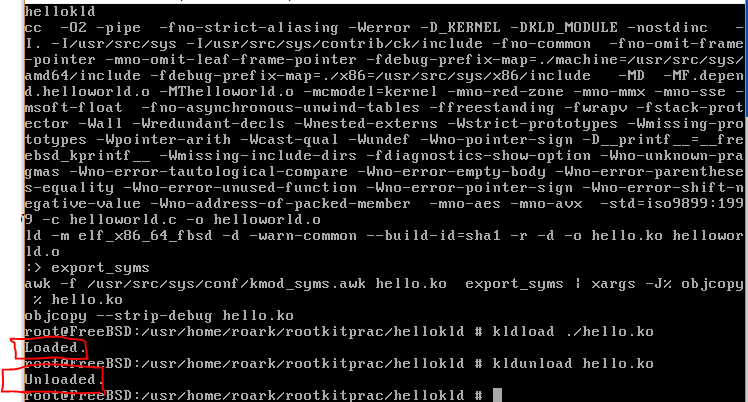
Now we have enough information to complete the basic KLD. To call the DECLARE\_MODULE macro we need to define the "data" section. We can just create a new moduledata\_t struct as such

static moduledata\_t hello\_mod = {  
"hello", /\* module name \*/  
load, /\* event handler \*/  
NULL /\* extra data \*/  
};

Then we can call DECLARE\_MODULE like this:

DECLARE\_MODULE(hello, hello\_mod, SI\_SUB\_DRIVERS, SI\_ORDER\_MIDDLE)&semi&semi;

Using the samba server, this file was moved to the vm and a Makefile was created and run. This produced a .ko file which could be run with the kldload and kldunload utilities on FreeBSD. This produced the following output.



This shows the KLD was successfully loaded and unloaded into/from the kernel.

### System Call Modules

This chapter was focused on System call modules, which are KLD's (learnt in the previous topic) that install system calls (system service requests), a mechanism an application uses to request service from the OS's kernel.

There are 3 items unique to each system call module:

1. **system call function**
2. ***sysent***structure
3. **offset value**

**System Call Function**

This part implements the system call, with its function prototype defined in sys/sysent.h:

*typdef int      sy\_call\_t(struct thread \*, void \*)*

thread \* is a pointer to the currently running thread and void \* is a pointer to the system call's arguments' structure, *if any.*

The system call function executes in kernel space, while the system call's arguments reside in user space.

**user space -**where all user-mode application run. Code running here cannot access kernel space directly. An application must issue a system call to access kernel space.

**kernel space -**where kernel and kernel extensions (KLD's) run. Code running here can directly access user space.

The kernel expects each system call argument to be of size *register\_t* (int/long depending on platform). It builds an array of *register\_t*values and then casts them to a void pointer and passes these as the arguments of the system call function.

**Sysent**

System calls are defined by the entries they have in a sysent structure, defined in the same header:

struct sysent {  
     int sy\_narg;                           /\* number of arguments \*/  
     sy\_call\_t \*sy\_call;                 /\* implementing function \*/  
     au\_event\_t sy\_auevent;      /\* audit event associated with system call \*/  
}

In FreeBSD the kernel's system call table is simply and array of these sysent structs, so whenever a system call is installed, its sysent structure is placed within the sysent array as such:

*extern struct sysent sysent[ ]*

**Offset value**

This is also known as the *system call number.*It is a unique number between 0 and 456 that is assigned to each system call to indicate its sysent structure’s offset within sysent[ ].

Within the system call module, the offset value must be declared:

*static int offset = NO\_SYSCALL*

No offset sets offset to the next available position in sysent[ ], however you could pick any unused number you want.

**The System Call Module Macro (SYSCALL\_MODULE)**

From the last chapter, it was learned that when a KLD is loaded, it must link and register with the kernel, and we used the DECLARE\_MODULE macro to do this. But when we write a SCM we use the SYSCALL\_MODULE, because it’s easier because it saves us the trouble of setting up the moduledata\_t data type. The module is defined as:

*#define SYSCALL\_MODULE(name, offset, new\_sysent, evh, arg)*

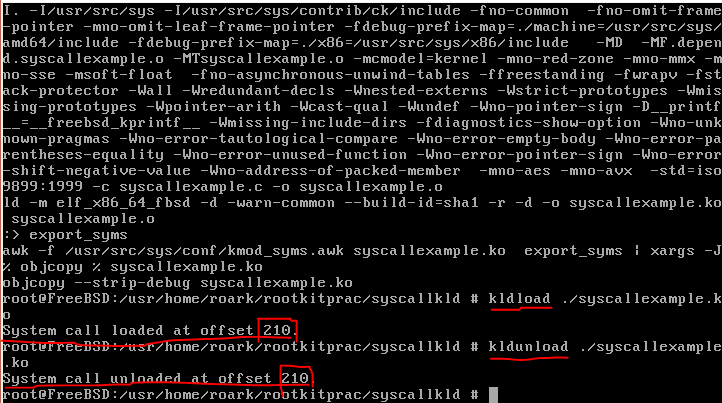
where:

* name
  + Generic module name **(passed as character)**
* offset
  + offset value **(passed as int pointer)**
* new\_sysent
  + completed sysent structure **(passed as struct sysent pointer)**
* **​**evh
  + event handler function (load/unload function)
* arg
  + arguments to be passed to evh. **For this project, it’ll always be NULL**

**RECAP**

To create the system call module we need

1. System call function and its arguments.
2. Sysent structure for the new system call.
3. the offset value (of sysent[ ]).
4. The load/unload function.
5. SYSCALL\_MODULE macro to link the KLD with the kernel.

4 was already done in the previous hello world/goodbye cruel world example, but it was appended to show where the system call module was loaded i.e. printing the offset value. Hence we just need to do 1,2,3 and 5. After doing this, and appending the given makefile, the following output was received.

We can see that when the system call module was loaded/unloaded, the relevant line was printed the terminal, with the offset value at which the system call was loaded/unloaded.

### Executing System Calls

Now that we have loaded the system call module into the kernel, we can execute the system call in 2 ways.

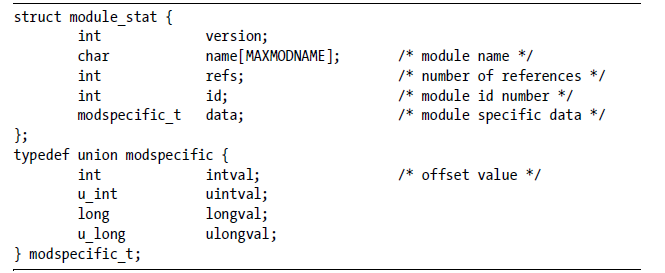
The first, is by the use of the syscall function, which executes a system call based on its system call number. This number is extracted using the modstat and modfind functions.

**Modfind**

The modfind function will return the *modid*of a kernel module, based on its module name. *Modid*is just a unique integer given to a module so that it can be identified by other functions, such as this one.

**Modstat**

The modstat function returns the *status*of a kernel module, referred to by its modid. The status information is stored in a structure of type *module\_stat*. Its fields are as such:



Within the module\_stat struct, is a modspecific\_t struct, and within this struct is our system call number, "*intval".* Hence the process to run the system call would be to:

1. Run modfind, given the modules name.
2. Run modstat, given the modid and an empty module\_stat struct
3. run syscall, given the stat.data.intval value, along with any input arguments that we want to put into the system call function.

This would all be within a simple C program.

There is, however, a second way to execute the system call. This is without the use of any C code. This is through the use of "perl" a programming language built into the command line. by running perl with the '-e' option, we can specify the command we want perl to execute. There is no C file being written, however just a simple line of code as such:

*perl -e '$str = "random string"; ' -e' 'syscall(syscall\_num, $str);'*

It can be seen that the " ' " (apostrophe) is used to denote a line to execute, and must be preceded by the -e option. sycall\_num can be obtained by simply noting the offset number at which the kld is loaded, after the kldload command is run.

### Kernel-User Space Transitions

This section focuses on a set of core functions that can be used from kernel space to copy, manipulate and even overwrite data stored in user space. We must recall the difference between user space and kernel space:

* User space - where all user-mode applications are run. Code running here ***cannot*** directly access kernel space.
* Kernel space - where KLD's run. Core running her ***can***access user space directly.

**Copyin and Copyinstr**

These functions allow the copying of a continuous region of data from user space to kernel space. The *copyin*function will copy *len*bytes of data from address *uaddr*to address *kaddr*:

*copyin(const void \*uaddr, void \*kaddr, size\_t len)&semi;*

The *copyinstr*function is similar to *copyin*except it copies a null-terminated ('\0') string, with the number of bytes successfully copied returned in *done*.

*copyinstr(const void \*uaddr, void \*kaddr, size\_t len, size\_t \*done)&semi;*

***Copyout***

This function is the direct opposite of copyin, it copies data from kernel space to user space:

*copyout(const void \*kaddr, void \*uaddr, size\_t len);*

***Copystr***

This is similar to copyinstr, but here the string is copied from one kernel space address to another:

*copystr(const void \*kfaddr, void \*kdaddr, size\_t len, size\_t \*done)&semi;*

That was really it for this chapter, just 4 core functions that will be made use of later on.

### Character Device Modules

Character Device Modules are KLD's that create/install a *character device*. This is just a fancy name for an interface that FreeBSD OS uses to access a device in the kernel. E.g. data is read from/written to the **system console** from the console, which is a**character device.**

For each character device module, there are 3 key items:

* *cdevsw*structure
* character device functions
* device registration routine

**"cdevsw" Structure**

Each character device is defined by the fields of its *device switch table,*i.e. the struct cdevsw, defined in <sys/conf.h>. There are many fields in this struct, and not all of them need to be set. The ones that are left null, are assumed as unsupported. The most relevant ones were found to be the following:

|  |  |
| --- | --- |
| **Entry Point** | **Description** |
| d\_open | Opens a device for I/O Operation |
| d\_close | Closes a device |
| d\_read | Reads data from a device |
| d\_write | Writes data to a device |
| d\_ioctl | Performs an operation other than read or write |
| d\_poll | Polls a device to see if there is data to be read/space available for writing |

There are 2 fields that must be defined in every one of these structures: d\_version, which indicates the versions of FreeBSD that the driver supports, and d\_name, the device's name.

**Character Device Functions**

For every single entry point defined in the cdevsw structure, a corresponding function must be defined. We must note that d\_version, however, does not need a definition. There is a function prototype for each entry point defined in <sys/conf.h>.

**The Device Registration Routine**

This process is just the method by which the created character device, located in the /dev directory, registers itself with the device file system (a.k.a DEVFS). This is done by calling the *make\_dev*function within the KLD event handler. This is done by:

* In the MOD\_LOAD case of the event handler function, create an instance of the character device by calling the make\_dev function with the parameters: &cdevsw\_character\_device, 0, UID\_ROOT, GID\_WHEEL, 0600, "character\_device\_name".
* In the MOD\_UNLOAD function, call the destroy\_dev function on the instance of the character device created in the MOD\_LOAD case as such: destroy\_dev(character\_dev\_instance).

**Recap**

So to recap, the steps to create the character device are:

* Declare the character devices entry points.
* Appropriately fill out the cdevsw structure.
* Implement each entry points function.
* Define the event handler function.
* **Use the DEV\_MODULE macro to instantiate the character device: DEV\_MODULE(name, event\_handler\_function, NULL)&semi;**

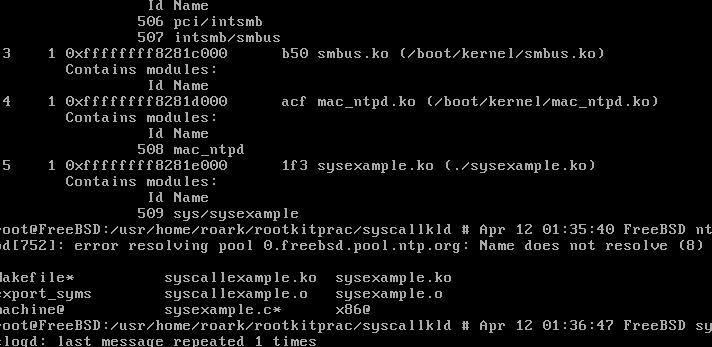
**Testing the Character Device**

To test a character device that has been created, we can write a simple C program that does the following

* define an instance of the character device prior to the main function: e.g. #define device "cd\_name", where cd\_name is the chosen character device name
* Call the functions that need to be tested by simply calling the function with the argument as the defined name above (device).

### Linker Files

The main point of note for this section was that when we use kldload/kldunload commands, the arguments we give to those are actually linker files, not the actual modules. We can see this by running the kldstat command with the -v option. It gives us a more in-depth look of any dynamically linked modules currently in the kernel:



We can see that 2nd loaded module (name did not fit) actually contains 2 other modules. These are the actual modules that are loaded into the kernel. What we write in the kldload/unload arguments are just the linker files. This means **that for every module loaded into the kernel, there is an accompanying linker file.**

## Chapter 2 – Hooking

### What is Hooking?

**Hooking**is when a handler function is created to modify control flow. This means that a "hook" function is created and its address is registered as the location for a specific function. So when that function is called the "hook" is run instead. In order to preserve the original behavior, the original function will be called at some point during the hook’s execution.

In terms of rootkit design, hooking is used to alter the results of the operating systems application programming interfaces (API's). Most commonly the results of bookkeeping and reporting API's can be changed.

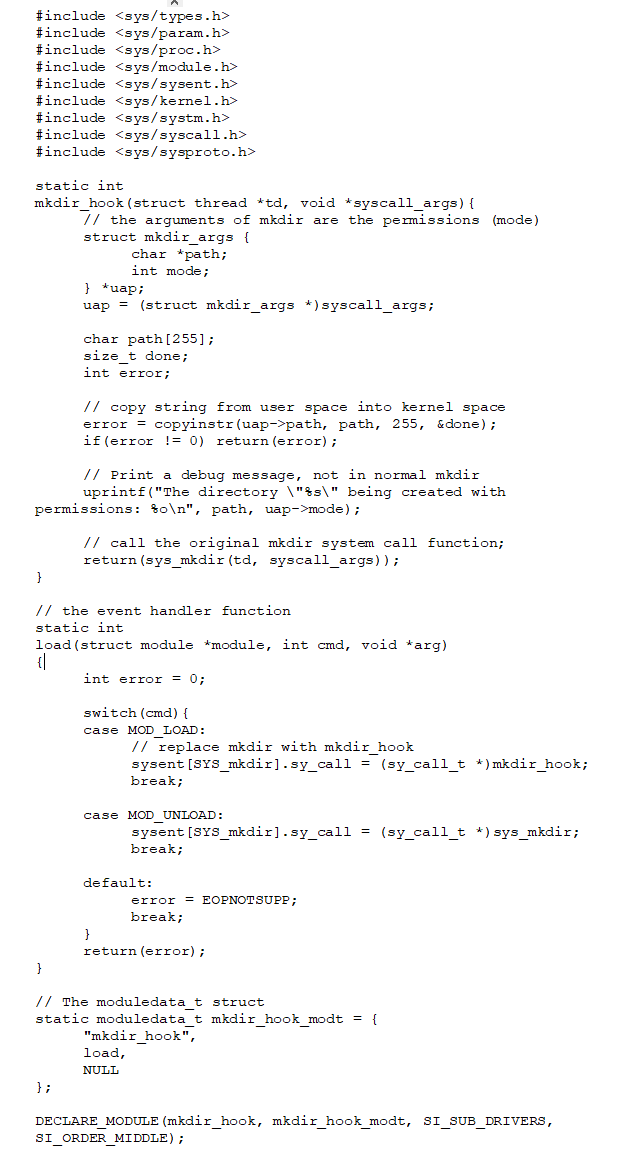
**Hooking a System Call**

If we are able to hook a system call, which is an entry point from which an application requests service from the OS's kernel, **we can alter the data the kernel returns to any or every user space process.**

In FreeBSD, this is done by registering its address as the system call function, which is within the system calls' sysent structure (the implementing function field).

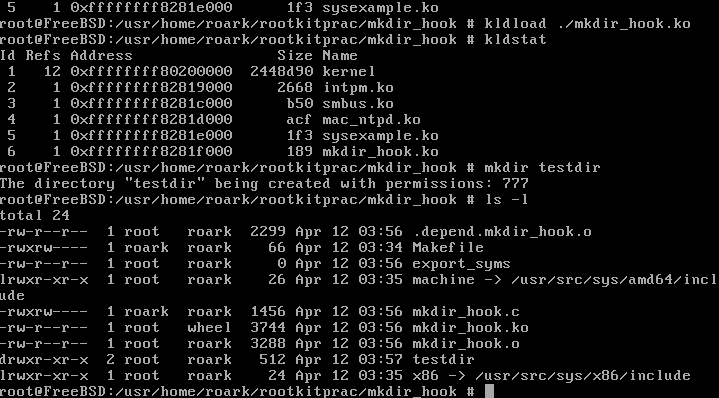
**Example Hook (mkdir)**

We could hook the mkdir system call, which creates a new directory, to do something other than just create the directory. I worked with the example of printing a message to the command line, saying the directory was created. Since we are overwriting the implementing function for the already created mkdir offset value, we only need to create the system call function, as the sysent struct is already there. All we need to do is overwrite sysent[SYS\_mkdir].sy\_call, which is the current implementing function of mkdir, with our implementing function. So, the code would be:

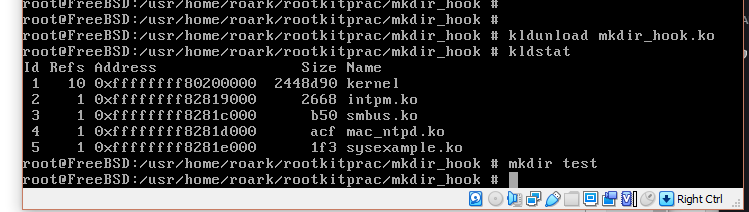


notice how we use sys\_mkdir, instead of mkdir. This is because the mkdir function cannot be used in a kernel module, since it is part of the standard C library which can only be used in user space programs. Since this is a KLD, we must use sys\_mkdir to denote that we are using the system call for mkdir. This annotation was only introduced in 2011, as the book was made in 2007 this was not explained in the book.

When we load the module, we get the following output:



Which is exactly as we expected. We can see that the kld is in fact loaded, from kldstat, and once a directory is created, the message is displayed, with the permissions as well. We can also see that the directory exists (testdir). When the kld is unloaded and mkdir is run, we instead get:



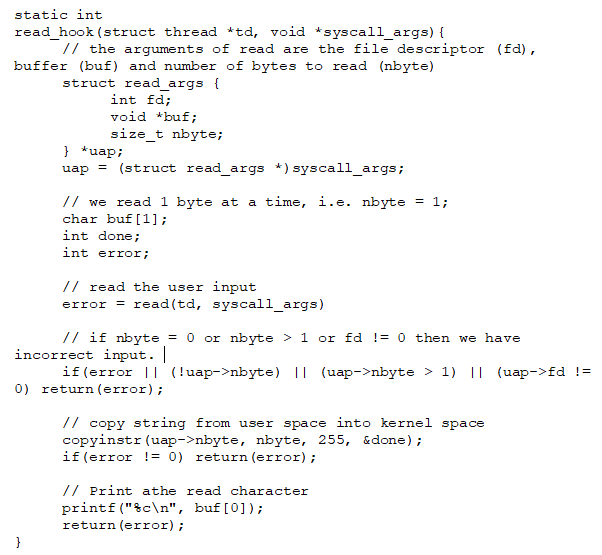
We see that no message is displayed, showing that the original mkdir system call function was loaded back into its proper place in the sysent table.

### Keystroke Logging

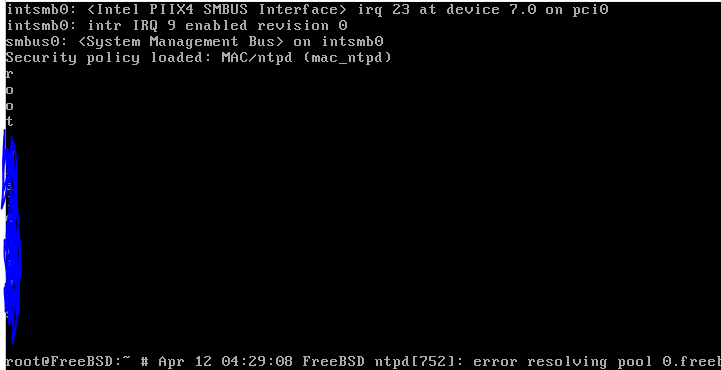
We can accomplish keystroke logging by hooking the read system call, since this call reads input. The C library definition is:

*read(int fd, void \*buf, size\_t nbytes)&semi;*

It reads nbytes of data from the object referenced by a file descriptor (fd) into a buffer (buf). So, to capture the user’s keystrokes, we just have to save the contents of buf before returning. The hooked function would look like this:



When this is run, along with the event handler and everything else a proper system call needs, we can proceed to testing it. I then thought about where the read system call is run, and the first place that came to mind was logging in. After logging in, and then printing the kernel buffer, using dmseg, we get:

You can see that the root username and password are printed there. I can now successfully make a keylogger in FreeBSD kernel.

### Kernel Process Tracing

Since I dont know enough about kernels to know which system calls to hook, there is a process I could use called *kernel process tracing, to figure out which hooks to implement.*

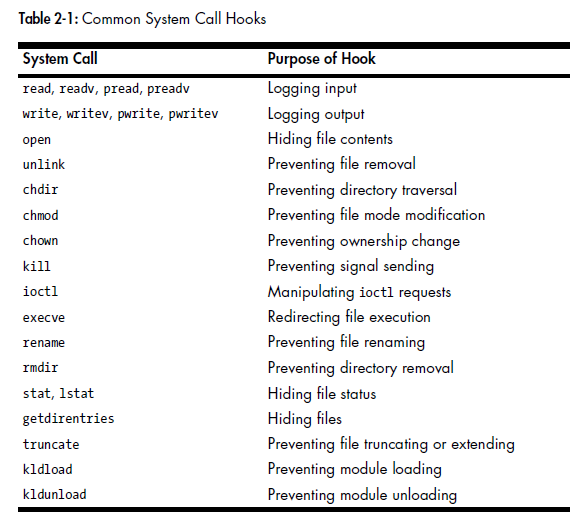
It serves as a debugging technique to intercept and record each kernel operation i.e. every system call, I/O signal processed and **namei translation.**This is done with the *ktrace*and *kdump*utilities.

**namei translation** is the following of a given path until a symbolic link is found. Symbolic links are special types of files that serve as a reference to another file or directory.

ktrace will enable the kernel trace logging for a specific process. e.g. ktrace ls will create debug files for the ls command. We then run kdump to display the data in the created trace files. From here we can see the various system calls that the command employs, hooking any of these calls will affect the operation/output of the command.

Hence the main point to realise is that when I know what I want to alter, but I dont know which system call to hook, I just need to perform a kernel trace on the system call.

I've attached some common system call hooks just for reference, when need be:



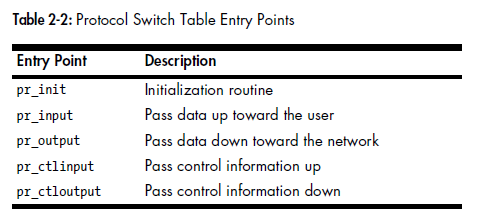
**Communication Protocols**

In FreeBSD, these are defined by their entries in a protocol switch table, examples of communication protocols are TCP/IP i.e. a set of rules/conventions used by two processes. **A rootkit can alter the data sent/received by either communication endpoint.**The entries of the protocol switch table is defined in a *protosw*structure, similar to the system calls being defined in a sysent structure. And even more similarly, these protosw structures are found within an *inetsw[]*array, similar to the sysent[] array. **Therefore in order to modify a communication protocol, we must go through inetsw[].**

Data passed between the two communicating process is stored in a buffer structure called ***mbuf***. To be able to read/modify this data, there are two main fields that should be noted:

* m\_len - specifies the amount of data contained within mbuf
* m\_data a pointer to the data

**Hooking a communication protocol**

****

The procedure for hooking a communication protocol is largely similar to hooking a system call. We need a hook function, and then load it properly in our event handler function. What we need to do is redirect the pr\_input entry point from the original function to our hooked function. For example assume that we want to print a debug message to the console whenever a certain phrase is encountered in an ICMP message. From COMP3331, I know that ICMP is at the network layer and hence must be encapsulated within an IP datagram. So within the mbuf struct lies the IP datagram data, since this is the data that is transmitted between the 2 communicating processes. To extract the ICMP message:

* We first remove the header length from m\_len.
* Then use the mtod function to get the ICMP message.
* Then we check if the icmp\_code field of this ICMP message is our certain phase.
* If so, then we print our success message, if not then just process the packet as normal, using icmp\_input.
* In the event handler function
  + in the MOD\_LOAD case, we find the ICMP protosw struct in the inetsw[] and change the pr\_input value to our hook function as such:
  + inetsw[ip\_protox[IPPROTO\_ICMP]].pr\_input = the\_hook\_function
  + *ip\_protox[IPPROTO\_ICMP] is defined as the offset, within inetsw[] for the ICMP switch table.*
  + in the MOD\_UNLOAD CASE, we set this value to the original icmp\_input function, which is the original function for processing the ICMP packet.
* Then we create our moduledata\_t struct and use the DECLARE\_MODULE macro as normal.

This concludes the concepts on hooking.

**Conclusion**

Hooking is really all about the redirection of function pointers. When a module is loaded, we want it to use our hooking function, whenever a system call is made, whereas when it is unloaded, we want it to return to normal execution.

## Chapter 3 – Direct Kernel Object Manipulation (DKOM)

### What is DKOM?

**What is Direct Kernel Object Manipulation (DKOM)?**

All OS's store their record keeping data within main memory. This is in the form of *objects*i.e. structures, queues etc. So when you ask the kernel for a list of running process, open ports, modules running etc, this data is parsed and returned. Since the data is stored in main memory, **it can be manipulated directly. There is no need to install a call hook to redirect the flow of control.**This is what this chapter is about.

**How is kernel data stored in FreeBSD?**

A lot of this data is stored as a *queue data structure*e.g. as:

* singly-linked lists
* singly-linked tail queues
* doubly linked lists
* doubly linked tail queues

They are defined in **<sys/queue.h>**along with 61 macros for operating on these 4 structures. I will show 5 macros for the *doubly linked lists*, since the macros for the other 3 queues are identical anyway. The 5 macros are:

* **LIST\_HEAD:**This is a structure that just contains one pointer to the first element of the list. If this struct is declared as LIST\_HEAD(HEADNAME, TYPE) head, where type is the data types of the elements in the list, then a pointer to headname can be declared as struct HEADNAME \*ptr.
* **LIST\_HEAD\_INITIALIZER:**The head is initialized by this macro as such: #define LIST\_HEAD\_INITIALIZER(head) { NULL }.
* **LIST\_ENTRY:**This structure connects the elements in the list. It is defined as:
  + # define LIST\_ENTRY(type)
  + struct {
    - struct type \*le\_next;
    - struct type \*le\_prev;
  + }
* **LIST\_FOREACH:**This struct allows traversal of the list. It contains a for loop, starting at LIST\_FIRST, and traversing using LIST\_NEXT.
* **LIST\_REMOVE:**this removes an element from the list (decoupled).

**Synchronization Issues**

It’s possible that while these lists are being traversed/modified, another piece of code tries to access/manipulate the same lists. This will cause data corruption. Running my code on one CPU, while the other process is on another thread in another CPU will still cause the same problem, because both are manipulating the same object. To safely manipulate kernel queue data structure, we need to acquire the appropriate **lock**first. So when our code is manipulating this data, nothing else can access it.

**mtx\_lock**

*Mutexes*are what allow this mutual exclusion of data. This is how thread sync is done. A thread acquires a *mutex* by calling mtx\_lock. If another thread currently has the mutex, the thread will sleep until the mutex is available.

**mtx\_unlock**

This is how mutexes are released. It has the same argument as the mtx\_lock function: struct mtx \*mutex).

**sx\_slock and sx\_xlock**

These are *shared exclusive locks.*They are reader/writer locks that can be held across a sleep.

* Multiple threads can hold a shared lock, but only one can hold an exclusive lock.
* If one thread has the exclusive lock, then no other thread may hold the shared lock.
* sx\_slock = shared lock.
* sx\_xlock = exclusive lock.

**sx\_sunlock and sx\_xunlock**

These release shared/exclusive locks.

### Hiding Running Processes Part 1

Now that we know about the macros and functions associated with how the kernel keeps track of data, and we know about DKOM, we can learn to hide a running process from a user who attempts to debug their computer.

**The proc Structure**

In FreeBSD, the context of each process is maintained in a proc structure, defined in proc.h. There are a few fields in *struct proc* that need to be understood in other to hide a running process:

* LIST\_ENTRY(proc) p\_list - contains *linkage*pointers associated with proc struct. It is referenced during insertion, removal and traversal of the list.
* int p\_flag - process flags, set on a running process. All are defined in proc.h.
* enum {PRS\_NEW = 0, PRS\_NORMAL, PRS\_ZOMBIE} p\_state - represents the current process state. PRS\_NEW = newly born and completely uninitialized process. PRS\_NORMAL = "live" running process, PRS\_ZOMBIE = zombie process.
* pid\_t pid - process ID, 32-bit int value.
* LIST\_ENTRY(proc) p\_hash - not learnt yet
* struct mtx p\_mtx - resource access control. It defines 2 macros, PROC\_LOCK and PROC\_UNLOCK for easily acquiring/releasing this lock.
* struct vmspace \*p\_vmspace - the vm state of the process.
* char p\_comm[MAXCOMLEN + 1] - executes the process. The MAXCOMLEN constant is defined in param.h (19).

**The allproc List**

FreeBSD has 2 lists for its’ proc structures. All process in ZOMBIE state are in *zombproc*while the rest are in *allproc.*It is referenced by *ps*and *top*commands. Thus you can hide a running process by simple removing its proc structure from the allproc list.

But this got me thinking, wouldn’t the process just not run if you delete its proc structure? Since there is no p\_comm entry, the process would not be able to execute. But it turns out the because processes are executed at *thread granularity,*modifying the process is not complicated.

allproc is defined as: **extern struct proclist allproc.**As this is defined as a proclist structure, and proclist is defined as: **LIST\_HEAD(proclist, proc),**we can see that allproc is just a kernel doubly linked list queue data structure, contained proc structures. The resource access control for the allproc list is defined as: **extern struct sx allproc\_lock.**

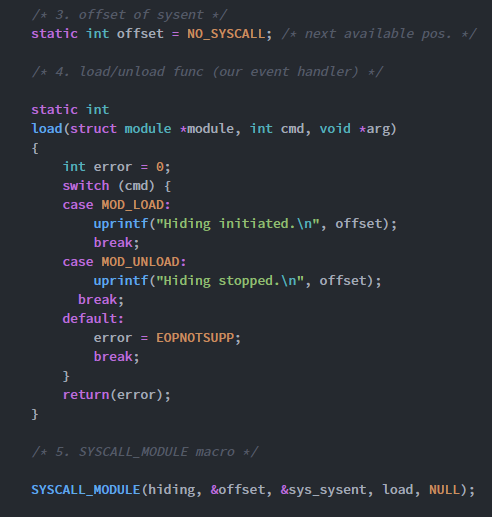
**Trying it out**

To try this out, I used knowledge learnt previous of system call modules and what was just explained, on the doubly linked list macros, locks and proc/allproc structs. So the steps I took to create this system call were:

* Create the system call function, with arguments being the process name, as a char ptr.
  + First we need to acquire the lock on the allproc list
  + Then iterate through the process list, and acquire a lock on each process as we check a number of things:
    - We check to see if the process virtual address space exists (proc->p\_vmspace), if true then unlock and continue
    - We check to see if the process flag is set to working on exiting, if true then unlock and continue
    - We check to see if the proc->p\_comm value is our process name (char ptr), if true, then we remove it from the process list, unlock and continue
  + Once iteration is done, we unlock the allproc resource and return.
* Then we create the sysent structure, offset and event handler functions.
* Finally collate it altogether using the SYSCALL\_MODULE.

Here is the code I wrote to test this out:

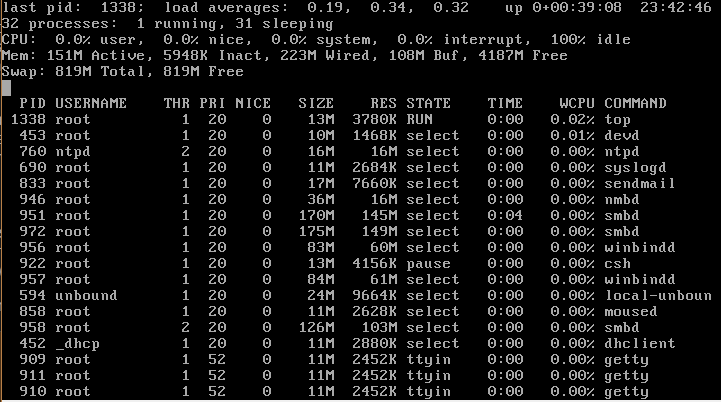




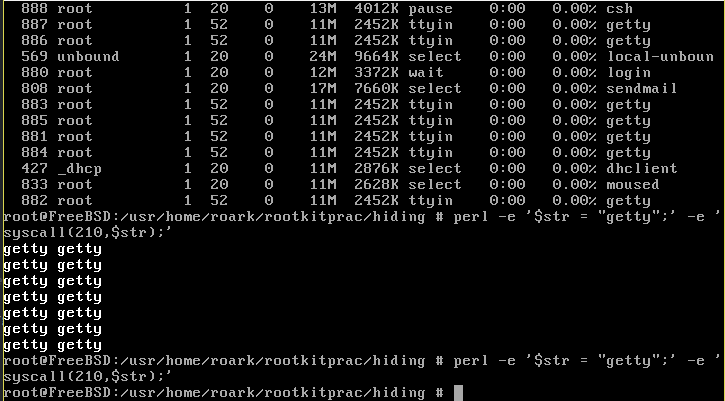
I have included comments to explain what I am doing. When the code is run in FreeBSD, we can test it out as such, again using perl because it’s just easier:

I ran the following perl command: **perl -e '$str = "getty";' -e 'syscall(210, $str); '**

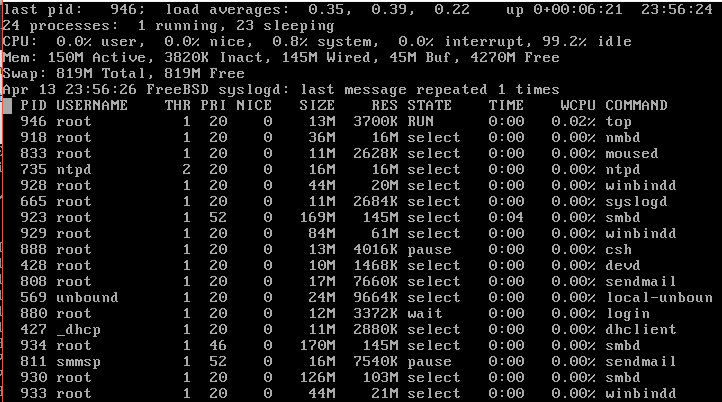
Since the output of top was:



I chose to hide getty. Running the command gives:

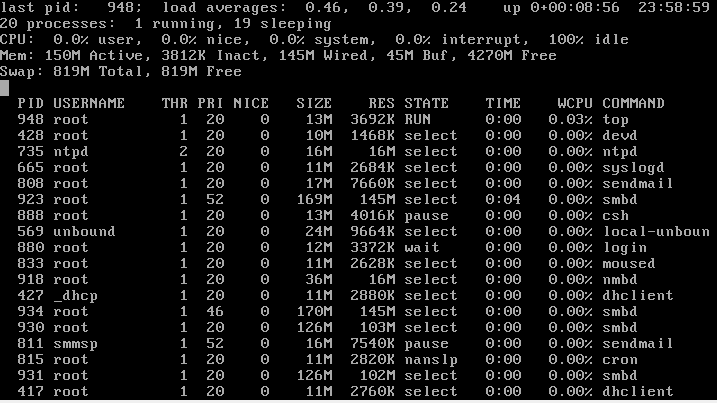


I appended my code to print my string and the process name whenever a match was encountered. So according to this, all 7 getty processes should now be hidden. Running top again shows:



We can see that there are no longer any getty processes. I wanted to try it one more time, with the winbindd process (used for samba). So, i ran the command: **perl -e '$str = "winbindd”;' -e 'syscall(210, $str);'**

The output of top after running this is:



We can see that the processes are hidden.

From this I have learnt that once I get a trojan process working, I know how to hide it on the victims computer. The rootkit will be doing this part.

### Hiding Running Processes Part 2

As I expected, hiding a running process turned out to be more than just manipulating the allproc list. A process can be found by it PID too. e.g. running **ps -p (insert PID here).**This way, even if it’s not there on the allproc list, ps still keeps track of it in this way.

To fix this, I need to learn about hash tables

**Hash Tables**

These are data structures in which keys are mapped to array positions by a hash function. It provides quick and efficient data retrieval. The key is transformed into a number that represents an offset in the array. Then you just retrieve the offset in O(1) time to get the value. It is what is used to locate a proc struct by its PID. It takes a lot less time than the O(n) for loop used to traverse the allproc list. The PID hash table is defined in proc.h as:

**extern LIST\_HEAD(pidhashhead, proc) \*pidhashtbl;**

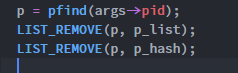
and initialized as:

pidhashtbl = hashinit(maxproc / 4, M\_PROC, &pidhash);

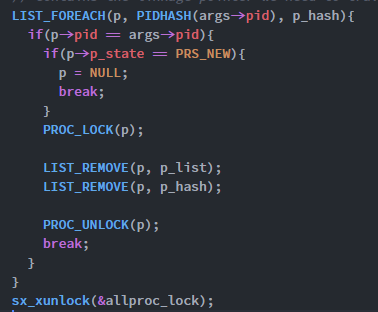
**The pfind Function**

To locate a process from the pidhashtbl, the kernel thread uses *pfind.*The allproc\_lock resource access control is also used for pidhashtbl, because the allproc list and pidhashtbl are designed to be in sync with each other. The hash table is traversed using the PIDHASH macro, which only takes in a PID, i.e. this is **the hash function**.

We can use this function to remove by PID instead of removing by name. It will be much faster, and only requires minimal changes to my previous program. We first need to change the input arguments to **pid\_t**type. From there, my first though was all I need to do is remove the LIST\_FOREACH part and replace with pfind. Then use the returned proc struct to remove p from plist and***p\_hash*** (which is the linkage for the hash table, similar to p\_list. I had marked this as not learnt in Pt 1. but it makes sense now). This is what I am talking about:



But this won’t work because the pfind function returns with the process being unlocked, therefore we need to put the LIST\_REMOVE macros inside the pfind function somehow. I just rewrote the function, as such:



Alternatively, I could just add the line **LIST\_REMOVE(p, p\_hash)** in my previous code, right under LIST\_REMOVE(p, p\_list). Removing the value that a PID maps to will remove the entire entry from the hash table, therefore that process cannot be found by running ps -p pid. This way I dont need to change the input arguments, they can stay as a char pointer.

Once I made these changes and ran the code, I attempted to test for the process smbd (another samba process). I ran perl -e 'syscall(210, 923)', after confirming that smbd had a process ID of 923. After running top I found that smbd with ID 923 was not there (there were other 923 processes however). Trying to kill the process using kill 493 gave the error that there was no such process. So the process has been deleted from both the hash table and the approc list, even though the process is still running.

**Using the exit function**

I've learnt that hiding an object with DKOM is difficult mainly because we have to remove all possible references to the object. But there is a function that already does this. The **exit**function. To fully hide the process I need to be able to patch:

* The parent process' child list.
* The parent process' process group list.
* The nprocs variable.

### Hiding an Open TCP Based Port

This part of the book was about hiding a TCP-based connection. To learn about this i first needed to learn about Internet protocol data structures.

**The inpcb Structure**

For each TCP/UDP socket, an *inpcb*struct, defined in netinet/in\_pcb.h, is created (the Internet protocol control block) to hold internetworking data such as:

* Network addresses
* port numbers
* routing information etc.

These are the field in inpcb that I need to understand in order to **hide an open TCP-based port.**This will become useful for transmitting data across the internet from the victims computer to my computer. Most of this is familiar from studying COMP3331.

* **LIST\_ENTRY(inpcb) inp\_list -** linkage pointer for the inpcb struct (used for insert, remove, and traversal of list).
* **struct in\_conninfo inp\_inc -**maintains the socket pair 4-tuple value in an established connection (src and dest ports and IPs):
  + IP addr
  + local port
  + foreign IP
  + foreign port
  + The above 4 data points are stored inside in\_conninfo, within a field called **struct in\_endpoints inc\_ie.**
* **u\_char inp\_vflag -**identifies the IP version in use and IP flags set on inpcb struct.
* **struct mtx inp\_mtx -**similar to proc struct, this is the resource access control for the inpcb struct, use INP\_LOCK and INP\_UNLOCK.

**The tcbinfo.listhead List**

The list of inpcb structs (doubly linked) is private to the TCP control module. It is called tcbinfo and defined as: **extern struct inpcbinfo tcbinfo.**This is quite a lot to take in at once, but looking for similarities between this and the proc struct/allproc list helps. We can see that tcbinfo is of type inpcbinfo. This struct has certain fields that must be understood to hide the TCP port:

* **struct inpcbhead \*listhead -**this is the main list, it points to the entire list starting at the head.
* **struct mtx ipi\_mtx**- this is the resource access control for the whole struct, similar to allproc\_lock. We use INP\_INFO\_WLOCK and INP\_INFO\_WUNLOCK.

Therefore we come to the conclusion that in order to hide an open TCP-based port, all we need to do is remove its **inpcb** struct from the **tcbinfo.listhead** list.

We can do this with a system call, with arguments taking in the local port. Then we iterate through the tcbinfo.listhead list, and look for when the local port of the incpb struct we find is that of our input. If we find a match, just remove it from the list. There were some problems encountered in coding this, because the book is outdated (made in 2007), some of the .h files have been updated. These are the changes I made:

* The in\_pcb.h file no longer has a macro for INP\_LOCK and the mtx\_lock and unlock functions are not used. Instead INP\_RLOCK and INP\_RUNLOCK macros are used, which use rw\_rlock and rw\_runlock functions
* The queues for the tcbinfo inpcb are not traversable using LIST\_FOREACH. A new type of queue called ck\_queue was used (it stands for concurrency kit, but I dont know too much about it). So wherever LIST\_FOREACH was used needed to be replaced with CK\_LIST\_FOREACH, and wherever LIST\_REMOVE was used, CK\_LIST\_REMOVE needed to be used.
* The listhead field of tcbinfo was changed to ipi\_listhead, so i just needed to change the name.

Once these changes were made the code compiled without errors. Below is my code:



I have omitted the sysent structure and the event handler function because they are exactly the same implementation as previous programs.

### Corrupting Kernel Data

An interesting though is what happens when one of the objects that I have hidden is found and killed? If for some reason the kernel decides to do nothing, then all is good. But what if the kernel instead goes and tries to remove the object from its lists and data? The object has already been removed, so it will not be able to delete anything. When the kernel fails to find that data in its lists, it will end up corrupting those data structures in the process, causing all sorts of problems to the kernel and OS. To prevent this we can:

* Hook the terminating function/s e.g. exit etc. to prevent them from removing the objects we have hidden
* Hook the terminating function/s to place the hidden objects back onto the lists before termination.
* Implement my own exit function to safely remove the hidden objects
* Do nothing. If the objects are never going to be found what’s the point of implementing safety procedures for their termination?

Since DKOM can only manipulate objects in main memory, it does have limitations, but there are still many objects that can be patched. These can be checked by executing them with the grep -r option in the terminal.

## Chapter 4 – Kernel Object Hooking

### What is Kernel Object Hooking

Kernel Object Hooking describes the hooking of objects that have entry points to the kernel, similar to the hooking of functions that have entry points, see [**Hooking**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/HookingARootkitTechnique/).

**Hooking a Kernel Object (e.g. A Character Device)**

A character device is an example of a kernel object that can be hooked. By modifying the devices' entries in the character device switch table, we can modify the behaviour of the character device itself. To do this we need to delve deeper into the cdev's themselves.

**The cdevp\_list Tail Queue and dev\_priv Structures**

In FreeBSD, all active cdevs are maintained on a private doubly linked tail queue named **cdevp\_list**. It is composed of **cdev\_priv** structures, linked together in Tail Queue fashion. In the cdev\_priv struct these are the fields of interest:

* TAILQ\_ENTRY(cdev\_priv) cdp\_list - The list of cdevp\_list itself. We use this to insert, remove and traverse the list.
* struct cdev cdp\_c - this contains the context of the cdev. The relevant fields of this are:
  + char \*si\_name - name of the cdev
  + struct cdevsw \*si\_devsw - pointer to this cdev's switch table

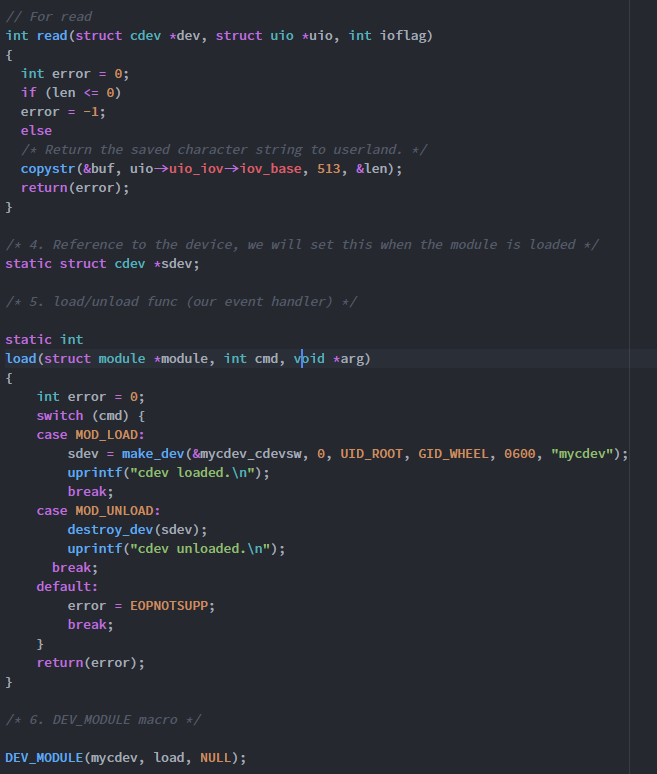
**The devmtx Mutex**

The resource access control for the cdevp\_list is: **extern struct mtx devmtx.**We use this for locking/unlocking the list.

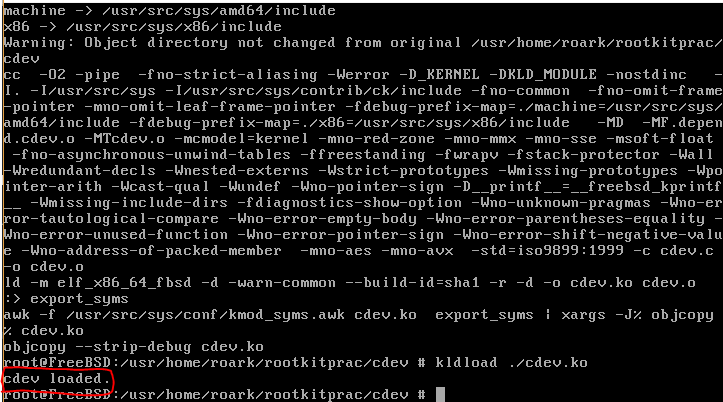
**Putting it together**

To complete this, we must first create an example character device and the testing file. Then we will hook its character device functions. To do this refer to the steps outlined [**here**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/CharacterDeviceModulesKldS/). After following those steps we arrive at the following code for the **character device creation**:





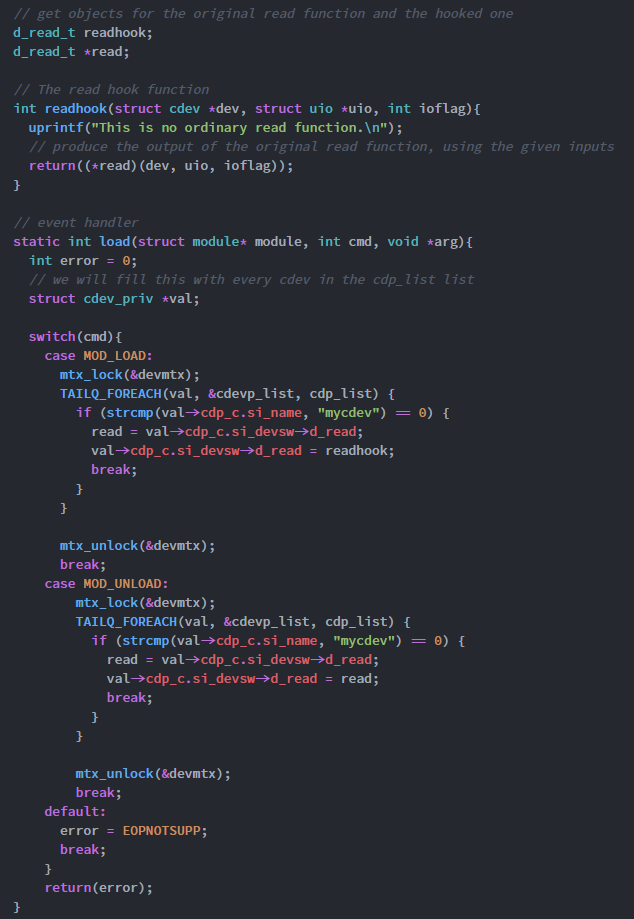
Compiling and loading the file gives the following output:



So we can see that the cdev was successfully loaded. Now if we want to hook this object, we would want to hook its entry points corresponding functions. Let’s say we want to hook the read function to not only do the original read, but also print a message to output. All we need to do is:

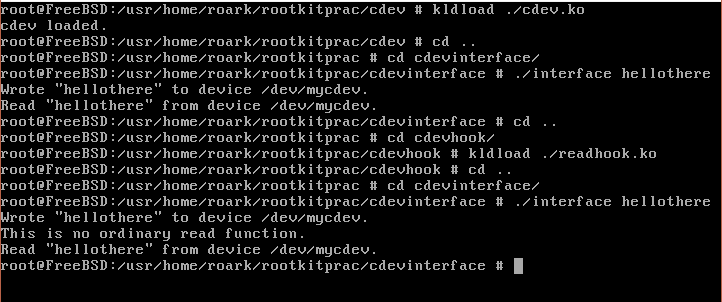
* Traverse the cdp\_list (making sure that we lock the resource first, using **devmtx**), until we find the cdev with the name we are looking for. **That is dev->cdp\_c.si\_name = the loaded device name.**
* When we find this, we need to go into its switch table and change the read functions its currently pointing to, to our one.
* At this time we also need to save the current read function, so that when this hooking KLD is unloaded, the original read function is restored.
* Then we unlock the devmtx resource

Putting this altogether, we get the following code:



The rest of the code is as normal for a DECLARE\_MODULE macro, we create the moduledata\_t struct and then fill in the macro. We can see that my hooked read file just prints "This is no ordinary read function", and then proceeds to read the file.

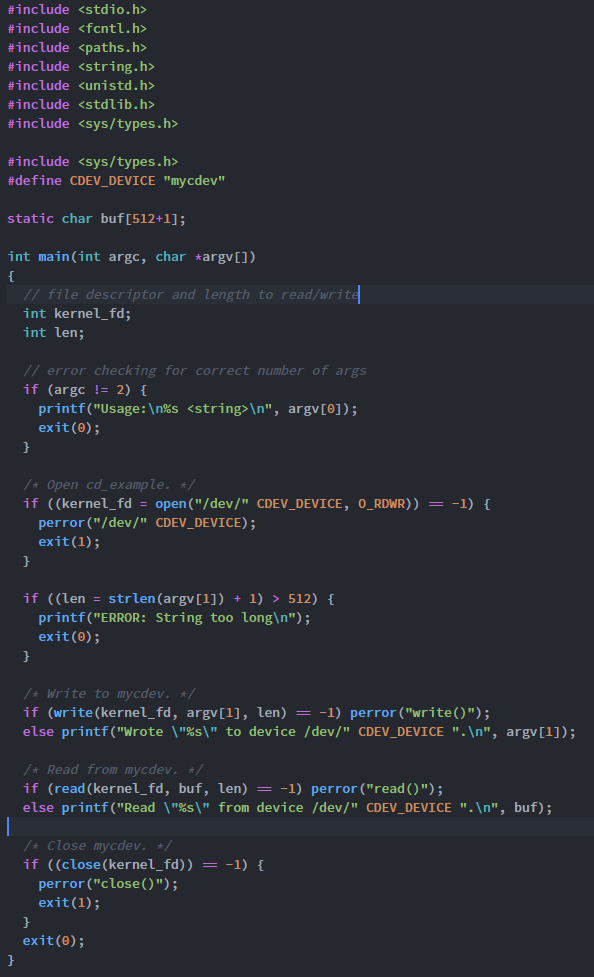
The creation of the character testing file is outlined [**here**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/CreatingCharacterDeviceTestingFile/). Once this was complete I was simply able to execute the testing file, with an input, after loading the hooking kld, and able to see the output as:



We can see that when the character device is loaded and the test file is run, we get the what was written and what was read printed back to us. But once the hook is loaded, we get the extra line "This is no ordinary read function".

### Creating a CDEV Testing File

The coding of the file was fairly easy. I followed the steps I had previously outlined [**here**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/CharacterDeviceModulesKldS/) under "Testing a Character Device". Doing so produced the following code:



To create the executable file I encountered a number of problems. First of all, I was attempting to use the previously used Makefile (that was used for KLD's) to compile the c file. This was giving errors that certain header files couldnt be found. So i tried finding a way around this by re-routing the header files (errors were recieved for stdio.h, stdlib.h, fcntl.h, string.h and paths.h) by prefixing them with ../../inlcude/. This is because the current working directory (cwd) was found to be /sys/. When we run cd .. from that folder we weirdly arrive at usr/src. From here, we need to navigate to usr and then include, hence why I use .. twice. but this still gave numerous errors.

After a few hours of debugging I realised that I wasn't compiling the code as a c program. I was treating it as a KLD. I needed to use FreeBSD's c compiler, cc, to compile it. So after running the code:

**cc -o interface interface.c**

I was then able to arrive at an executable file to give input to my character device.

## Chapter 5 – Run-time Kernel Memory Patching

### Introduction to Run-time Kernel Memory Patching

In this chapter we move away from KLD's (introducing code into a running kernel), and into how to patch and augment the kernel with userland code. This is done by interacting with **/dev/kmem, a device that allows reading and writing to kernel VM, i.e. kmem allows patching of various code bytes (code loaded in executable memory space) that control the logic of the kernel.**This is what Runtime Kernel Memory Patching is.

**Kernel Data Access Library (libkvm)**

This is the interface through which we can access the kernel vm (**kvm**), through using the kmem device of course. There are 6 function from libkvm we need to use:

* **kvm\_openfiles -**This is how we initialize access to kvm. If this call is successful, a descriptor is returned, which is used in all subsequent libkvm calls. If there is an error, NULL is returned. It has the prototype:
  + **kvm\_t \* kvm\_openfiles(const char \*execfile, const char \*corefile, const char \*swapfile, int flags, char \*errbuf)**
  + execfile - The kernel image to be examined (must contain*symbol table,*not explained yet). If set to NULL, the currently running kernel image is examined.
  + corefile - The kernel memory device file must be set to
    - /dev/mem (set to NULL)
    - crash dump core generated from savecore
  + swapfile - not used, always set to NULL
  + flags - indicates the r/w permissions of the file
    - O\_RDONLY
    - O\_WRONLY
    - O\_RDWR
  + errbuf
    - if the function encounters an error, the error message is written here.
* **kvm\_nlist**- gets the *symbol table*entries from a kernel image. Has prototype:
  + **int kvm\_list(kvm\_t, \*kd, nlist \*nl)**
  + nl - null terminated array of **nlist**structs. The fields of **nlist**are
    - n\_name - name of symbol in memory
    - n\_value - address of symbol
  + kd - descriptor from kvm\_openfiles
  + This function iterates through nl, looking up each symbol in the n\_name field in the current kernel image. If found, the n\_value field is filled out, otherwise it is set to 0.
* **kvm\_geterr -**Returns a string describing the most recent error condition on the kvm descriptor (\*kd). Its prototype -> **char \* kvm\_geterr(kvm\_t \*kd).**
* **kvm\_read -**Reads data from kvm. If successful, returns the number of bytes transferred, otherwise returns -1. Prototype is:
  + **ssize\_t kvm\_read(kvm\_t \*kd, unsigned long addr, void \*buf, size\_t nbytes)**
  + similar to most read functions, read into buf, nbytes at addr.
* **kvm\_write -**Writes data to kvm. returns nbytes, if error then returns -1. Prototype:
  + ssize\_t kvm\_write(kvm\_t \*kd, unsigned long addr, const void \*buf, size\_t nbytes)
  + if successful, it will return nbytes, which is number of bytes written to addr from buf.
* **kvm\_close -**closes the open kvm descriptor (kd). **int kvm\_close(kvm\_t \*kd),**if successful returns 0, otherwise, -1.

### Allocating Kernel Memory

First a brief rundown of x86 call statements:

**What are they?**

In x86, call statements are used to call a function/procedure. There are two types, near and far, but I only need to focus on **near call statements.**When a call statement is reached, the address of the instruction after the call statement is saved on the **stack,**so that the procedure knows where to return to. Therefore the machine code operand in hex will be the address of the called procedure(i.e. if the statement is call addr1, it will be addr), minus the address saved on the stack, e.g. addr2, i.e. addr1 - addr2.

**Allocating Kernel Memory**

This section explains what you need to do if the patch you are applying is bigger than what was already there and will overwrite nearby instructions. We need to then allocate kernel memory. These are the core functions

* **malloc -**allocates a number of bytes in kernel space
  + if successful, returns the kernel virtual addr, otherwise returns NULL
  + prototype: **malloc(unsigned long size, struct malloc\_type \*type, int flags)**
    - **size -**amount to allocate
    - **type -**used for stats, but normally set to M\_TEMP
    - **flags**
      * **M\_ZERO -**0 allocated memory
      * **M\_NOWAIT -**if the request can’t be fulfilled immediately, return fail (NULL). Used for interrupts.
      * **M\_WAITOK -**will sleep and wait for resources if can’t allocate immediately. **Cannot return NULL if this is set.**
* **MALLOC Macro -**calls the malloc function, defined as:
  + **MALLOC(space, cast, unsigned long size, struct malloc\_type \*type, int flags)**
* **free -**deallocates memory that was allocated by malloc:
  + **free(void \*addr, struct malloc\_type \*type);**
  + addr is the return value of malloc call, type is the malloc\_type from malloc
* **FREE MACRO -**calls free function, has prototype
  + **​FREE(void \*addr, struct malloc\_type \*type)**

**Allocating from User Space**

Here is where we actually use run-time kernel memory patching i.e. allocate space from user space. What we need to do is defined by an algorithm:

* Retrieve the memory address of mkdir
* save sizeof(kmalloc) bytes of mkdir, we need the kmalloc routine (kernel memory allocation)
* overwrite mkdir with kmalloc
* call mkdir
* restore mkdir

Basically this process changes a system call to execute your code instead and then restores the original system call. It’s like what a KLD would do except you aren’t using a KLD. **I have to remember that when the system call is overwritten, any process that uses or is currently using the call will break, resulting in kernel panic.**

**Putting it together**

So remembering that the 6 libkvm function can be run from userspace, we can use them to compile a C program in userland. Also with the previous kmalloc program, we had that the 3 instructions that require dynamic linking were the M\_TEMP, COPYOUT and MALLOC locations. They had locations of 10, 34 and 64 respectively within the kmalloc program example. So the steps we need to take are:

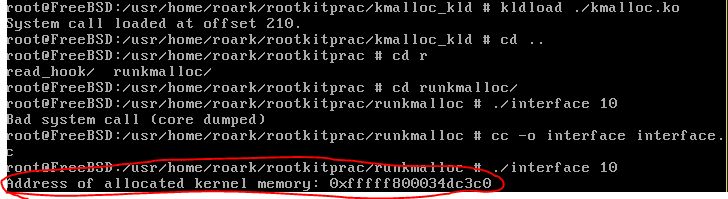
* Get the descriptor, kd.
* Find the address values for mkdir, M\_TEMP, malloc and copyout.
* Patch the kmalloc function code to contain the addresses for M\_TEMP, malloc and copy out, at the right places
  + use the knowledge of call statements, so the call statements address - the addr of the statement directly after
* Save the sizeof(kmalloc) bytes of mkdir.
* Overwrite mkdir with kmalloc.
* Allocate the kernel memory.
* restore mkdir.
* close the descriptor, kd.

Following the steps above gives us the following code:





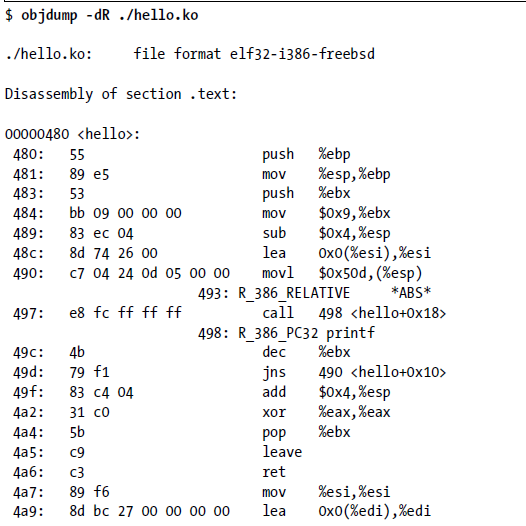
The kmalloc char array was found by running the objdump with the -d option on the previously created kmalloc.c code. This gave us the code which we would patch. The final output comes to:



Showing that memory was successfully allocated from user space using the libkvm functions.

### Patching Code Bytes

Now that we know the functions that can be used to patch kvm, we can start issuing system calls that patch code bytes in this kvm. I was shown an example system call whose function was to just print out a message 10 times, using a for loop. Code bytes are written as assembly code, which is familiar to me because of my study on COMP2121. So patching certain bits out can be very dangerous if I don't know what the code does first. That’s why the book says to run the objdump command with the -dR option on the example KLD to first understand the assembly text.



We can see that at address 49d, the instruction executed is: **jns 490 <hello+0x10.**This is saying jump back to 490 if the sign flag is not set. This forms the for loop. **If we perform a nop instruction on this, then the system call will act differently, only printing the message once**. We can do this using the functions from the previous sections, *note that the name of the kld is "hello", so this is the symbol we will be looking for*:

* First we need to make an nlist struct initialised with "hello" for the name and NULL for the value.
  + This is because we want to find the address of hello, and then patch it with the NOP instead of the 0x10 instruction.
* Then we initialise memory access using**kvm\_openfiles**. We are using the current image, we want to use dev/mem and we want to read and write, so we use:
  + **kvm\_t \*kd = kvm\_openfiles(NULL, NULL, NULL, O\_RDWR, errbuf)**, where errbuf is just a char array.
* Then we run the **kvm\_nlist** function using kd and nlist struct we made. This will set the value for hello. This will be the address of the first piece of code in the hello kld.
* Now, we need to save this and search through it until we find the jns instruction (defined by 0x79), then we replace this with the nop instruction (defined by \0x90).
  + To save it, we use**kvm\_read**, the descriptor is **kd**, the **addr** is **n1[0].n\_value** (because we want the value of hello), the **buffer** is any buffer we make, and the **size** should be total size of the kld, which is about **0x29** bytes (4a9 - 480 = 0x29 = **41 bytes**).
  + Then we search through the buffer looking for when an element of the buffer = 0x79. We save this as our **offset.**
* Now we write into kvm, using **kvm\_write**at the position defined by the start of the hello kld, **n1[0].n\_value,**and add the offset. Here we write in our **nop code (\0x90\0x90, because we want to write two nops).**
  + **​**The descriptor is again **kd,**the addr is n1[0].n\_value + offset
  + The const buffer is our nop code and the size is the size of the nopcode buffer.
* The final step is then to just close the kd, using **kvm\_close.**
* NOTE THAT ERROR CHECKING FOR EACH OF THE FUNCTIONS OUTPUTS IS NECESSARY.

### Inline Function Hooking

This is pretty neat type of hooking that takes advantage of assembly code. It places an **unconditional jump instruction**within the body of a function to a region of memory under my control. This part will contain:

* the new code I want the function to execute.
* the code bytes that were overwritten by the unconditional jump.
* an unconditional jump back to the original function.

But If I wanted to I dont have to do the last part. I could just not include and unconditional jump back to the original function.

Using the given example, I attempted to use inline function hooking to patch the mkdir system call so that it will output a phrase every time a new directory was created. To do this, we must first look at the disassembly code of mkdir, using **nm**command as such:

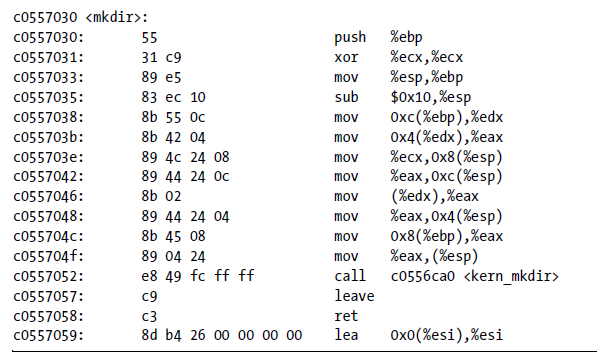
**nm /boot/kernel/kernel | grep mkdir**

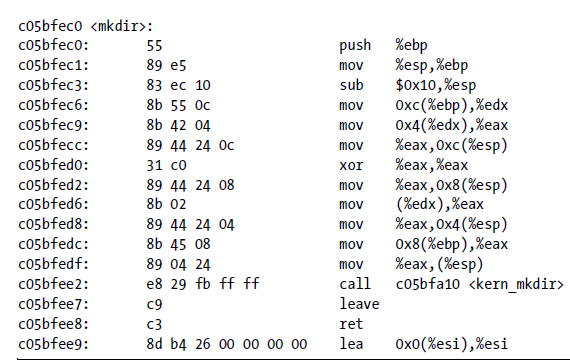
Running the above code will produce the set of instructions that the mkdir system call uses. Looking at this we can figure out where to place the jump, which bytes to preserve and where to jump back to. I will be studying the given example, since recalling from my previous post, the mkdir command cannot be used in the kernel anymore, it required sys\_mkdir. But since this sys\_mkdir function also uses mkdir, it is too difficult to figure out which instructions are solely for sys\_mkdir, and which ones are shared between the two.

It’s also important to know that the disassembly of a system call is different for every machine. So, it’s not possible to make an inline function hook that works for every machine. Each system call that you are hooking, or function in general, must be disassembled and studied.

**Applying the hook**

So, to apply the hook, we need to make 7 bytes of space, because this is the amount that the unconditional jump requires. But we also need to figure out where to jump back to. From the two examples given below:





We can see that both make use of the kern\_mkdir function. This is where we can jump back to, but it also means we must save every byte up until this byte. This means everything until 0xe8 (the first byte of the kern\_mkdir instruction). So, the steps to apply the hook now combine what we've previously learned about malloc, and the use of the libkvm functions. The steps we need to take are:

* Initialize the descriptor
* Find the addresses of mkdir, M\_TEMP, malloc, copyout and print (like our code of kmalloc from before
* search through mkdir until we find 0xe8, and save this as our offset
* The amount of memory we need will be the size of our string (prints the string using uprintf) **+**the size of the preserved mkdir code (everything before 0xe8) + the size of the unconditional jump.
* We then patch kmalloc, to contain the M\_TEMP, malloc and copyout codes as before
* Allocate the kernel memory using kvm\_write to write kmalloc to mkdir's memory pointer
* Once memory has been allocated, restore mkdir
* Then we patch the string printing code with the uprintf code we found from using kvm\_nlist, and write this into memory.
* Next we put in the preserved mkdir code.
* Then finish off with the unconditional jump.

That concludes the teachings for Inline Function Hooking

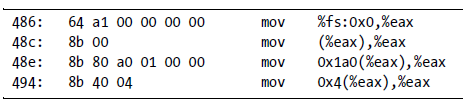
### Cloaking System Call Hooks

To my surprise, there seems to be a way to implement a system call hook, without patching the system call table or its function. This seems to be a very clean way to hook a function. It is achieved by patching the **system call dispatcher**within an inline function hook, so that it references our **hooked system call table** instead of the original.

In FreeBSD, the system call dispatched is **syscall.**So we can go and apply an inline function hook to where this function is implemented (in **/sys/i386/i386/trap.c**). In this code, there is a line that references the original system call table, and stores the address of the actual call in a sysent structure.We can disassemble this line, study it and produce an inline function hook to our own system table, but remember that every machine is different, so the same implementation won’t work for every machine.

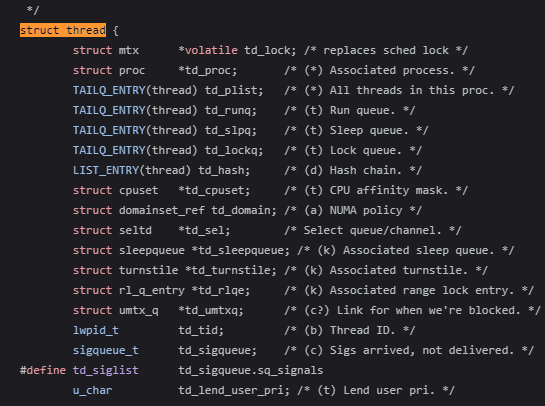
**Disassembled Call to the System Call Table**

If we see the example given:

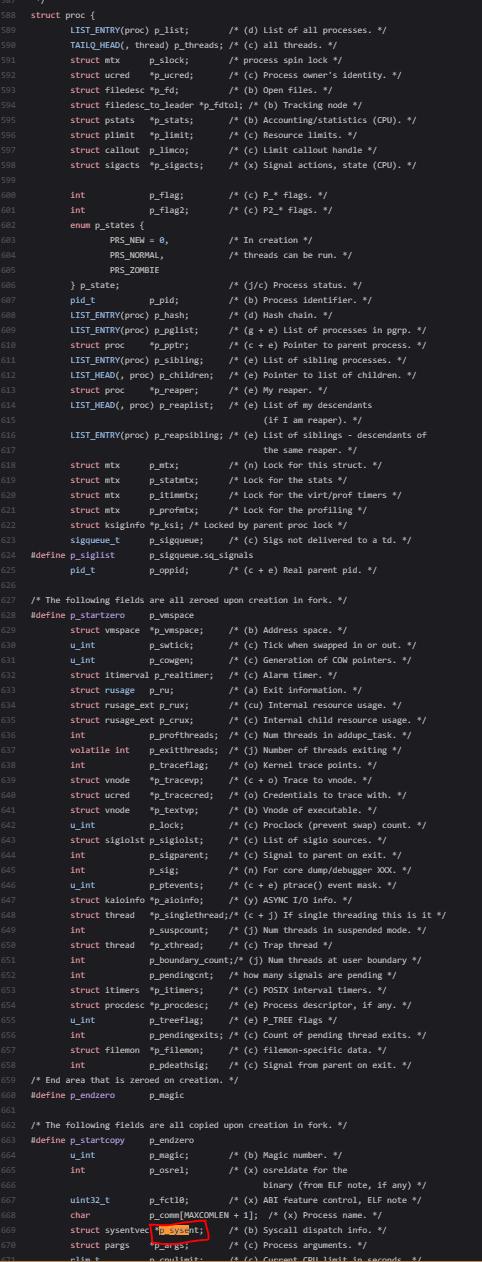




The first image is the disassembled code of the second image. We need to understand what the disassembled code is doing in order to hook it. The first line loads the currently running thread (defined by the %fs register) into register %eax. The thread is a very large structure, but I managed to get an image of the first few fields in it:



This seems to be an updated version of thread, hence the second line (which is meant to be loading the threads associated process into register %eax) should have the offset of struct mtx as well. Previously, the **struct proc \*td\_proc**line was the first entry in the thread struct. Now the next line is loading the 0x1a0 offset of the proc struct into %eax. Looking at the proc struct:



we can see that this is the Syscall dispatch info struct called p\_sysent. The very last instruction then loads the 0x4 offset within the sysentvec (the type of p\_sysent) into %eax. Looking at the sysentvec struct, this would be:



the second value, since int is a 4 byte value. Hence the pointer to sysent, called sv\_table is stored in %eax.

**What We Really Want to Do**

This is the line we want to change. Instead of pointing to the original sysent struct, we want it to point to our malicious sysent struct. After we do this, any system call modules we load will no longer work. All we need to do is patch these new modules, since we now control the system call for loading a module as well!

## Chapter 6 – Putting it All Together

### HIDSes

Finally, we arrive at our last step: putting together everything we've learned to make a complete rootkit. It will be used to bypass something knows as a **Host-based Intrusion Detection System (HIDS)**

**What is a HIDS**

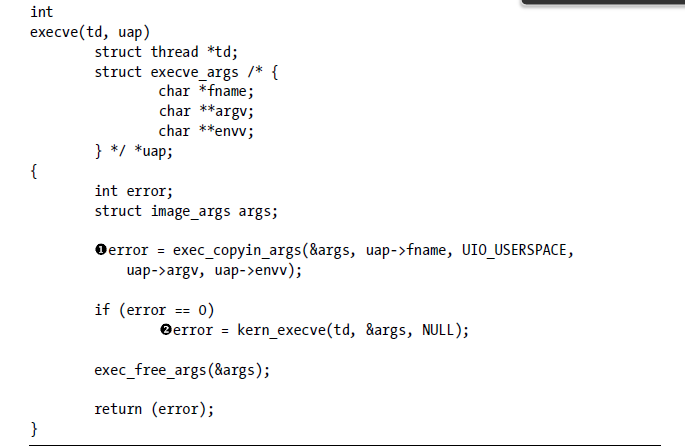
It just monitors, detects and logs modifications done to files on a filesystem. For every file the HIDS will create a crypto hash of the data and record it in a database. Any changes will of course change the hash and it will compare hashes during its audits. If the files differ then it is flagged.

**Bypassing HIDS**

The flaw with HIDS is that it trusts the system's APIs (programming interfaces). We can hook these APIs to get past HIDS.

**Execution Redirection**

We can do this through *execution redirection*which is the execution of one binary with another. For example say we have an original binary called original.c and a trojan binary trojan.c. All we do is intercept the request to execute hello.c and replace it with the execution of trojan.c. The HIDS will never pick this up because the **original binary is not changed.**

We can do this by hooking the execve system call, which is responsible for file execution. It is implemented in **/sys/kern/kern\_exec.c.**Examining this file we see two key lines: 

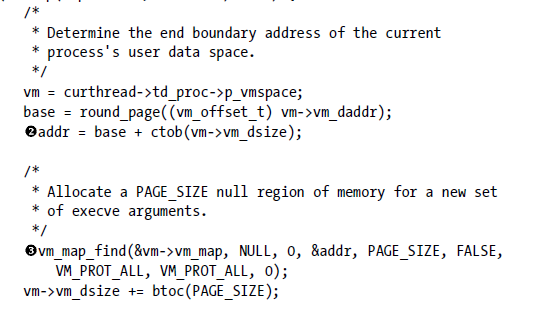
The first one just does a simple copying of the arguments from user space (entered in command line) to a buffer (args). The buffers contents are then passed to the real execution function **kern\_execve.**So executing a new binary just means rewriting the set of execve arguments recieved before the exec\_copyin\_args bit!

I was reluctant to try out the examples on my own computer because I dont have anything going wrong the FreeBSD virtual system I have, but I was able to identify the steps required:

* First, we check if the name (execve\_args->fname is the descriptor for the name of file) is equal to our original file (original.c)
* We want to allocate a region of memory for the new set of execve arguments (this process was rather difficult to understand so I explained it [**here**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/AllocatingForNewSetOfExecveArguments/)).
* Next we want to create the new execve\_args struct, with our trojans details
  + fname -> the name of the file
  + argv -> arguments to the file execution
  + envv -> environmental variables for the file
* We then insert this into the allocated user data space we just did.
* We then call execve on this new user data space instead of the old one.

We have tricked HIDS... sort of. The trojan file we made is still unrecognized to HIDS so it will be flagged. This is where File Hiding comes in.

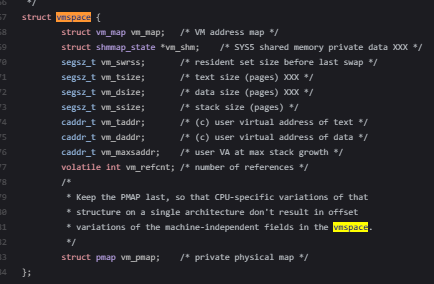
### Allocating Space using VMAP

This section of the execve hook was difficult to follow, since it included concepts I haven’t heard of before. Below is the section of code I tried to understand:

The main point here is to obtain the end boundary address, of the current process, so that we can make the new set of execve arguments there. But to do that, we need to also set this region to null. So:

**Step 1 - Get the Address**

To get the address we need to go inside the vmspace struct of the current process. This is stored in thread->proc->vmspace. Below is an image of the contents of that struct:



The address we want will be the current address of the process (vm\_daddr) + all the data that it has (vm\_dsize). When we compute this our equation will be:

**finaladdr = vmspace->vm\_d\_addr + vmspace->vm\_dsize.**

Since the dsize value is in pages, we must use the **ctob**function to transform from page to bytes.

**Step 2 - Allocate the Correct Size**

Now that we have the address, we want to allocate a region of this for the new arguments (trojan execve). To do this we use the **vm\_map\_find**function which will find a free region of memory in a map, and map an object to it. Going through the prototype, we can figure out what values we need:

**vm\_map\_find(vm\_map\_t map, vm\_object\_t object, vm\_ooffset\_t offset, vm\_offset\_t \*addr, vm\_size\_t length, int find\_space, vm\_prot\_t prot, vm\_prot\_t max, int cow)**

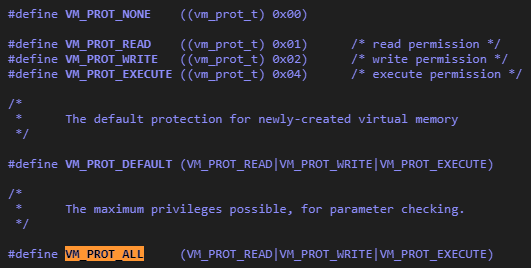
Our map is in vmspace->vm\_map, the object we are mapping is just NULL, we dont have an offset, the address is the finaladdr value calculated in step 1, the length of the data is just the size of a page which is defined at PAGE\_SIZE and is a constant (4096). The find\_space value  specifies the strategy to use when searching for the free region. Every value other than VMFS\_NO\_SPACE, **vm\_map\_findspace** is called to allocate the free region. That’s why it is set to false, because we want this function to run. The prot, max and cow values are passed to the vm\_map\_insert function.

This function has prototype:

**int vm\_map\_insert(vm\_map\_t map, vm\_object\_t object, vm\_ooffset\_t offset, vm\_offset\_t start, vm\_offset\_t end, vm\_prot\_t prot, vm\_prot\_t max, int cow)**

We can see from the underlined section that the prot, max and cow values are there. This is what each of them does in this insert function:

* prot - this defines the protection values for the space. We want to use VM\_PROT\_ALL because this gives it the permissions to read, write and execute from that space (image below)
* max - I am still unsure about this one
* cow - This indicates the flags that should be sent to the new entry. Since we dont have any flags, this is set to 0.



After doing this, we have now successfully allocated space for the new set of execve args.

### KLD and File Hiding

**File Hiding**

We can hide files by hooking the *getdirentries*system call to just not keep track of our file. Basically, this function will read in directory entries, referenced by a file descriptor, into a buffer. On success, the number of bytes successfully transferred is returned, otherwise a -1 is returned. The entries read into the buffer are stored as *dirent*structures which have their own definition. **The HIDS will be reading the buffer, so all we have to do is prevent the getdirentries system call from storing the dirent struct for our file.**To do this, we once again perform a hook, but on the getdirentries call. We can summarise the process into the following steps:

* First we call getdirentries on our input arguments (which is our file descriptor) if the return value is greater than 0, then we know that there are files to check.
* Then we have to check the contents of buf by **examining each dirent structure inside**and comparing the name of each current dirent structure with the name of our trojan file.
  + Remember we need to copy into kernel space first. (copyin function)
* If we find a match, then we simply remove the structure from the buffer, and then copy it back out into user space (copyout function).
* We then also adjust the number of bytes transferred so that it was like the file was really never there

But this only works because of the KLD that we have loaded (containing the hook). If someone just runs a simple kldstat, they will be able to see our KLD loaded there.

**KLD Hiding**

Here, the rootkit is the KLD. If we want to hide it we can just use DKOM, because when the KLD is loaded, **the linker file is really the one being loaded (which contains 1+ kernel modules).**These linker files and KLDs are stored on two different lists: *linker\_files*and *modules*. The first, *linker\_files*contains the set of loaded linker files, while *modules*has the set of loaded kernel modules. The hiding routine should then traverse these lists and remove the ones we want.

**The linker\_files List**

This file is defined as type ***linker\_file\_list\_t,*in /sys/kern/kern\_linker.c.**If we look up the definition of linker\_file\_list\_t, in sys/linker.h, we get:



So we can see that it is just a doubly linked tail queue made of *linker\_file*structs. **We need to search these linker\_file structs and remove the ones that will reveal us.**There are a couple interesting things about the linker\_files data type however:

* Whenever a linker\_file is loaded (i.e. added to the linker\_files list), the *next\_file\_id*number is given to the files ID number, and then the stat is incremented.
* There is no dedicated lock for this, as there was for other lists. Therefore we need to use a new type of lock called "Giant" which protects the entire kernel. It is defined as:
  + **extern struct mtx Giant**

**The linker\_file Structure**

The fields of interest to us are below (but full definition is in sys/linker.h anyway):

* int refs - The reference count for this file. **Important to note that the very first linker\_file is the kernel image itself. Whenever other linker files are loaded, its reference count is increased by 1. Therefore this value must be decremented to fully hide the linker file.**
* TAILQ\_ENTRY(linker\_file) link - this forms the list that we use to traverse, remove or insert linker\_files from.
* char\* filename - the linker file's name.

**The modules List**

Similar to the linker\_files list, this is also a doubly linked tail queue, but made of *module*structures. Also similar to linker\_files, it has a counter, *nextid*, just like next\_file\_id. This list does however have a dedicated lock, defined as **extern struct sx modules\_sx.**

**The module Structure**

The fields of interest are (defined in /sys/kern/kern\_module.c):

* TAILQ\_ENTRY(module) link - the list that we reference when inserting, removing and traversing the list.
* char\* name - the kernel module's name.

**Actually Hiding the KLD**

Now we put all this together, **remembering to decrement the kernel image reference count. the linker files counter(next\_file\_id) and the modules counter (nextid) all by one.**These are the steps we need to take:

* We need to reference the linker\_files list, the kld\_mtx lock, next\_file\_id, module list (doubly linker tail queue), nextid and the entire module struct, since none of these are defined in any header files.
* Since this is done at runtime, we can apply all these changes to the modules list and linker files list during the loading of this KLD. So we put all our code in the event handler function
* First, we acquire the locks on the linker\_files list and each KLD  (using mtx\_lock on Giant and kld\_mtx).
* Then we decrement the kernel images reference count, **using linker\_files->tqh\_first->refs, which points to the reference number of the first linker file (kernel image)**
* Then we iterate through the linker\_files list looking for the KLD we are writing.
  + Use TAILQ\_FOREACH to iterate, referencing **link** for the list, and an empty linker\_file structure to check each one.
  + If we find a match, then we decrement the next\_file\_id and remove the linker file from the list, using TAILQ\_REMOVE.
* Then we release the lock on the linker\_files list and KLD, using mtx\_unlock.
* Now we need to delete from the modules list, **so use sx\_xlock on the modules\_sx defined lock**
* Iterate through the modules list looking for the name of the .ko file we loaded
  + Use TAILQ\_FOREACH again, referencing the **link**list again and using an empty module structure (from the definition we had in the first step)
  + If we find a match, decrement nextid and remove the module from the list (using TAILQ\_REMOVE
* Then, finally, we remove the lock on the modules list.

After implementing these steps we can successfully hide the existence of the KLD and modules from the relevant lists. This however makes it unloadable, meaning it cannot be unloaded.

### File Updates

**Directory Changes**

Whenever a directory has files added/removed or any other changes done to it, its access and modification times change too. These can be checked to figure out that something is wrong. But as I figured out, these can changed too. Instead of keeping track of what’s going on and constantly updating the system, **we can just roll back the changes whenever we do something to a directory. This, however, can only be done to the access and modification times.**

**Using the stat function (Access and Modification Times)**

The stat function will return all the statistics data for a certain directory, and store it in a struct of type *stat*. This struct has many fields but the ones relevant here are:

* time\_t st\_atime - time of last access
* time\_t st\_mtime - time of last data modification

Once we have stored these times, we can then proceed to do whatever we want to the directory. We store these using a timeval struct, which has the following definition:

struct timeval {  
       long tv\_sec; /\* seconds \*/  
       suseconds\_t tv\_useci; /\* and microseconds \*/  
}&semi;

So we just need to create a two, timeval, element array, to store the values of st\_atime and st\_mtime, like this:

**struct timeval time[2]**

**time[0].tv\_sec = st\_atime**

**time[1].tv\_sec = st\_mtime**

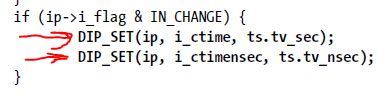
After this, we then do our deeds to the directory. When we are done, we call the function utimes, to set the values of the newly recieved st\_atime and st\_mtime (because of what we did to the directory) back to the values just prior to our malicious deeds:

**utimes("dir", (struct timeval \*)&time)**

**Change Times**

The change time value **cannot be set or rolled back.** But what we can do is stop the change time value from even being set. That means after a certain point, it just doesn't change.

To do this, we need to change the implementation of the function *ufs\_itimes*which is responsible for updating the change times for files. It works by converting a given vnode (which is a file, directory character device i.e. any object in kernel memory that **speaks the UNIX line interface which means that it uses read, write etc.)**into an inode which is a readable data structure. There are many checks done during the implementation of ufs\_itimes, but the most important one is done when there the ip->iflag value is set to IN\_CHANGE. This means that the file was changed. What we want to do is remove the implementation of this if statement by just inserting a nop instead:



This requires going through the assembly code of the ufs\_itimes and finding when those 2 exact lines are called. We just go and replace those 2 lines with 2 nops. I didn't go into the implementation because the assembly file was extremely difficult to go through.

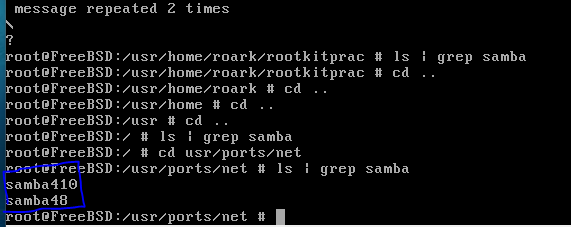
Extra Technical Research/Work

# FreeBSD and Samba Installation

In order to work with the FreeBSD OS, I had a couple options. The first was to create another boot partition off which my system could start up from, meaning every time i would want to run FreeBSD I would need to restart my computer. The other option was using a virtual machine software, such as VirtualBox to run a virtual image of the FreeBSD OS, which would be installed on a virtual disk image or virtual hard drive (.vdi or .vhd file). I decided to go with the second option because having both Windows and FreeBSD running allows me to accomplish more things at the same time, e.g. reading the book while putting the examples into practice.

I downloaded an ISO image of FreeBSD of their website, created a new virtual disk image in VirtualBox and mounted the ISO image onto the new vdi. Following the instructions to install on the FreeBSD website was easy enough as I ended up with with a working copy.

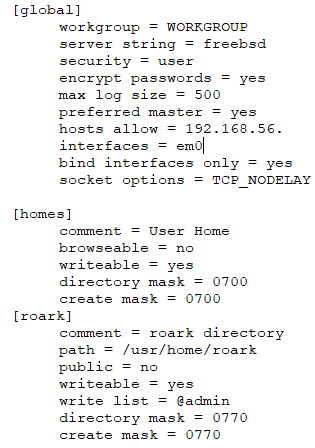
Now that I had the OS installed, the next step was to set up a form of text editing software and enable file sharing capabilities so I could transfer files from the Windows disk to the FreeBSD virtual disk. Getting a text editor was as simple as running a pkg install command (ee and vi) in the FreeBSD terminal, but setting up the filesharing server was a bit more difficult. Firstly, I searched up how to share files from a virtual BSD machine to the host computer, and the SAMBA tool came up. This linux based tool creates a server upon which the host computer and virtual computer can transfer files. To set this up I first navigated to the usr/ports/net directory which contains all the version of samba. I then ran the ls command by piping it through the grep command, searching for any matches to "samba". This way I can locate the laters version of samba on the OS and install it.



We can see that the latest version is samba48. Now, we can just navigate to this folder and run the "make config" and then "make install clean" commands. This starts a very large amount of installs of various packages. After this is done we can start setting up our server by editing these files:

1. /usr/local/etc/smb4.conf
2. /etc/rc.conf

The first one contains the server data and the folders that we want to share. Currently, I have only selected one folder to share, which is my user folder, named "roark" on the virtual machine. The files contents are as such:



The global settings define the servers characteristics, workgroup must match the workgroup of the computer, the server string is an arbritary value, security defines the the way users log on to the samba server. If they use the same user settings as the FreeBSD machine then this value should be set to user. The hosts allow value indicates the range of inet address that the server will accept connections from. Since the network adaptor has been configured as a virtual one and host-only, it will specifically only connect to this server so we dont need to worry about connections from other inet addresses. The home settings defines the directory of user profiles under /usr/home and the roark settings sections defines the exact folder that we want to share, with the path being the /usr/home/roark directory.

Once these settings are made, the next step is to enable the startup of the samba server. this is done by adding the line "samba\_server\_enable = "YES"" in the rc.conf file located in /etc. This allows us to run the samba\_server service without the use of the onestart command as such:

***service samba\_server start***

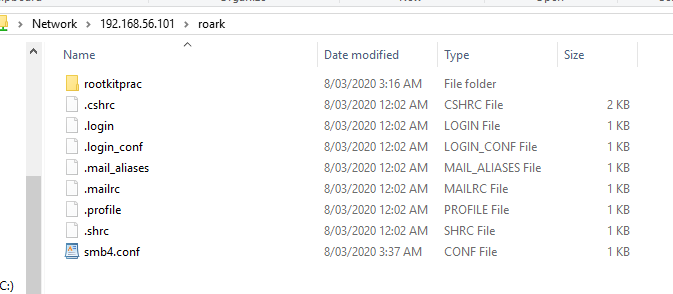
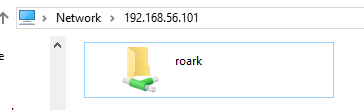
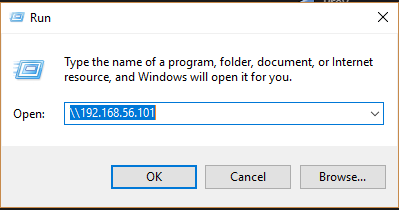
However, for some reason I have yet to identify, the OS isnt reading the rc.conf file properly and giving the error that "samba\_server\_enable = "YES"" could not be found. Hence our only other option is to use onestart instead of start as such:

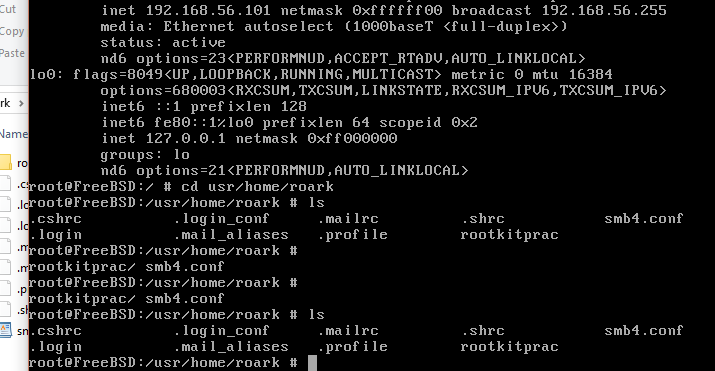
***service samba\_server onestart***

This command will start the samba tool which is comprised of 3 daemons (*background process not under direct control of interactive user*) called nmbd smbd and winbindd.

1. nmbd - this is a server process that undesrands and can reply to NetBIOS name service requests.
2. smbd - this process provides the file sharing/printing services to the windows client, which in our case is the host computer (my personal computer). It is also responsible for user authentication, resource locking and data sharing through the SMB protocol. It listens to default TCP ports 139 and 445.
3. winbdd - this service is actually controlled by the winbind service, not the samba service, but it resolves user and group information for the running samba server.

Now to figure out the inet address of the running server, we issue a ifconfig command and check the inet value. This was found to be  192.168.56.101. We can connect to this server in windows by copying this to the run dialog box, after inputting two forward slashes:





We can see that the server is running properly and any files added from the host side, show up on the virtual machine as well. The Samba file sharing tool has been successfully set up.

# Brief Introduction to Operating Systems Theory

In order to study rootkits, I first need to understand some basic operating systems theory. Luckily a quick Youtube search revealed a series of videos by a person called John Phillip Jones. Viewing his videos allowed me to gain a good understanding of operating systems, which I will detail below.

**Operating Systems Theory**

An operating system consists of a series of management modules that form a software which can be controlled by the user. There are 4 main managers that an operating system needs:

**Memory Manager**

There needs to be code to control the memory of the computer (RAM). It needs to keep track of where applications are in the memory. It is important to note that applications are not run off disk memory (HDD) or RAM. Instead these applications are copied to the RAM at run-time and their machine instructions are sent to the CPU (comprised of a control unit and an arithmetic/logic unit) which executes the instructions. The post-processing data is then saved by the memory unit (RAM). The CPU is what connects input devices (e.g. a mouse and keyboard) to output devices (e.g. monitor or printer).

For example let application 1, 2 and 3 be the names of applications already loaded in memory *(running applications are processes and idle ones are called programs).* This means that the machine code for these applications is loaded into "particular" locations in memory. Some kind of code needs to keep track of these locations (start address/end address) and this is what forms the memory manager.

**File Manager**

When a user clicks on data that is on the disk (HDD), the OS needs to know where exactly this data is in order to run the application. It usually does this by keeping a table which keeps track of these locations (whether they are the applications that are going to be run or the data that those applications use). The OS must also keep track of whether these files are read only, read/write or executable. This code that manages the files forms the file manager.

**Processor Manager (CPU Time)**

When there is more than 1 application in memory and the user is attempting to run them simultaneously, it will appear to this user that the 2 program are indeed running simultaneously. However this is not what is actually happening. In the CPU, machine instructions for one application is being executed for a certain amount of time, and then machine instructions for another application is being executed. Therefore certain amounts of time are given to each application to ensure that they are all being run together. This process is done by code, which manages the applications access to CPU time. This forms what is known as the processor manager.

**Device Manager**

When external devices are connected to the computer, the user must be notified and given a choice as to what should be done with the device. Resources must be allocated/deallocated for these devices. Therefore the OS needs code to manage these external devices and this is what is known as the device manager.

All of these above managers work together, each relaying information to the user interface.

**OS Purpose**

To control:

1. Every time step of processing time
2. Every section of main memory
3. Every file
4. Every device
5. *The access levels for certain users*(relevance to rootkits)

# Trojan vs Rootkit

**Trojan**

A trojan looks like a legitimate program on the outside, but a certain condition or action will trigger the trojan. The trojan could also hide within another program that appears to be trustworthy. These programs do not replicate, making them very unlike viruses or worms.

**Rootkit**

A rootkit is a program that hides in a victims computer, allowing someone at a remote location to take control of this computer. Once installed, a variety of things can be accomplished such as:

* Execution of programs
* Changing Settings
* Installing processes
* Hiding processes
* Monitoring activity
* Accessing files

It is a type of virus that gives root access to a hacker.

# My First System Call

This chapter was focused on System call modules, which are KLD's (learnt in the previous topic) that install system calls (system service requests), a mechanism an application uses to request service from the OS's kernel.

There are 3 items unique to each system call module:

1. **system call function**
2. ***sysent***structure
3. **offset value**

**System Call Function**

This part implements the system call, with its function prototype defined in sys/sysent.h:

*typdef int      sy\_call\_t(struct thread \*, void \*)*

thread \* is a pointer to the currently running thread and void \* is a pointer to the system call's arguments' structure, *if any.*

The system call function executes in kernel space, while the system call's arguments reside in user space.

**user space -**where all user-mode application run. Code running here cannot access kernel space directly. An application must issue a system call to access kernel space.

**kernel space -**where kernel and kernel extensions (KLD's) run. Code running here can directly access user space.

The kernel expects each system call argument to be of size *register\_t* (int/long depending on platform). It builds an array of *register\_t*values and then casts them to a void pointer and passes these as the arguments of the system call function.

**Sysent**

System calls are defined by the entries they have in a sysent structure, defined in the same header:

struct sysent {  
     int sy\_narg&semi&semi&semi&semi&semi;                           /\* number of arguments \*/  
     sy\_call\_t \*sy\_call&semi&semi&semi&semi&semi;                 /\* implementing function \*/  
     au\_event\_t sy\_auevent&semi&semi&semi&semi&semi;      /\* audit event associated with system call \*/  
}

In FreeBSD the kernel's system call table is simply and array of these sysent structs, so whenever a system call is installed, its sysent structure is placed within the sysent array as such:

*extern struct sysent sysent[ ]*

**Offset value**

This is also known as the *system call number.*It is a unique number between 0 and 456 that is assigned to eah system call to indicate its sysent strcuture's offset within sysent[ ].

Within the system call module, the offset value must be declared:

*static int offset = NO\_SYSCALL*

No offset sets offset to the next available position in sysent[ ], however you could pick any unused number you want.

**The System Call Module Macro (SYSCALL\_MODULE)**

From the last chapter, It was learned that when a KLD is loaded, it must link and register with the kernel, and we used the DECLARE\_MODULE macro to do this. But when we write a SCM we use the SYSCALL\_MODULE, because its easier because it saves us the trouble of setting up the moduledata\_t data type. The module is defined as:

*#define SYSCALL\_MODULE(name, offset, new\_sysent, evh, arg)*

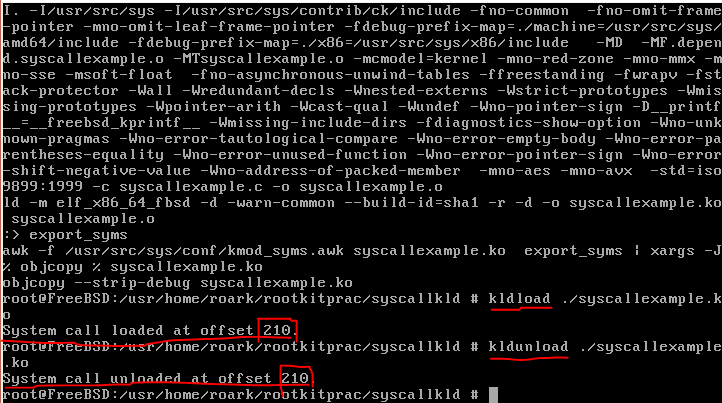
where:

* name
  + Generic module name **(passed as character)**
* offset
  + offset value **(passed as int pointer)**
* new\_sysent
  + completed sysent structure **(passed as struct sysent pointer)**
* **​**evh
  + event handler function (load/unload function)
* arg
  + arguments to be passed to evh. **For this project, itll always be NULL**

**RECAP**

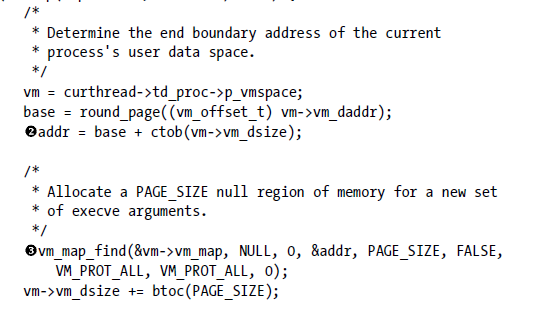
To create the system call module we need

1. System call function and its arguments.
2. Sysent structure for the new system call.
3. the offset value (of sysent[ ]).
4. The load/unload function.
5. SYSCALL\_MODULE macro to link the KLD with the kernel.

4 was already done in the previous hello world/goodbye cruel world example, but it was appended to show where the system call module was loaded i.e. printing the offset value. Hence we just need to do 1,2,3 and 5. After doing this, and appending the given makefile, the following output was recieved.

We can see that when the system call module was loaded/unloaded, the relevant line was printed the the terminal, with the offset value at which the system call was loaded/unloaded.

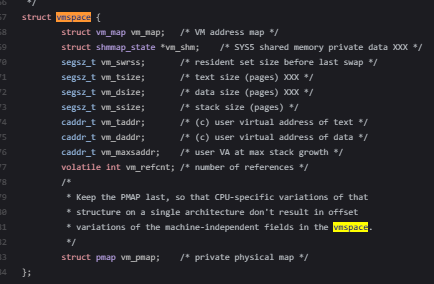
# How to Allocate Kernel Space

This section of the execve hook was difficult to follow, since it included concepts I havent heard of before. Below is the section of code I tried to understand:

The main point here is to obtain the end boundary address, of the current process, so that we can make the new set of execve arguments there. But to do that, we need to also set this region to null. So:

**Step 1 - Get the Address**

To get the address we need to go inside the vmspace struct of the current process. This is stored in thread->proc->vmspace. Below is an image of the contents of that struct:



The address we want will be the current address of the process (vm\_daddr) + all the data that it has (vm\_dsize). When we compute this our equation will be:

**finaladdr = vmspace->vm\_d\_addr + vmspace->vm\_dsize.**

Since the dsize value is in pages, we must use the **ctob**function to transform from page to bytes.

**Step 2 - Allocate the Correct Size**

Now that we have the address, we want to allocate a region of this for the new arguments (trojan execve). To do this we use the **vm\_map\_find**function which will find a free region of memory in a map, and map an object to it. Going through the prototype, we can figure out what values we need:

**vm\_map\_find(vm\_map\_t map, vm\_object\_t object, vm\_ooffset\_t offset, vm\_offset\_t \*addr, vm\_size\_t length, int find\_space, vm\_prot\_t prot, vm\_prot\_t max, int cow)**

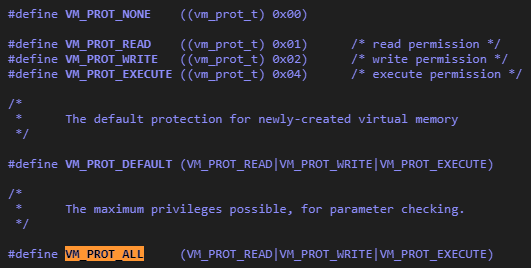
Our map is in vmspace->vm\_map, the object we are mapping is just NULL, we dont have an offset, the address is the finaladdr value calculated in step 1, the length of the data is just the size of a page which is defined at PAGE\_SIZE and is a constant (4096). The find\_space value  specifies the strategy to use when searching for the free region. Every value other than VMFS\_NO\_SPACE, **vm\_map\_findspace** is called to allocate the free region. Thats why it is set to false, because we want this function to run. The prot, max and cow values are passed to the vm\_map\_insert function.

This function has prototype:

**int vm\_map\_insert(vm\_map\_t map, vm\_object\_t object, vm\_ooffset\_t offset, vm\_offset\_t start, vm\_offset\_t end, vm\_prot\_t prot, vm\_prot\_t max, int cow)**

We can see from the underlined section that the prot, max and cow values are there. This is what each of them does in this insert function:

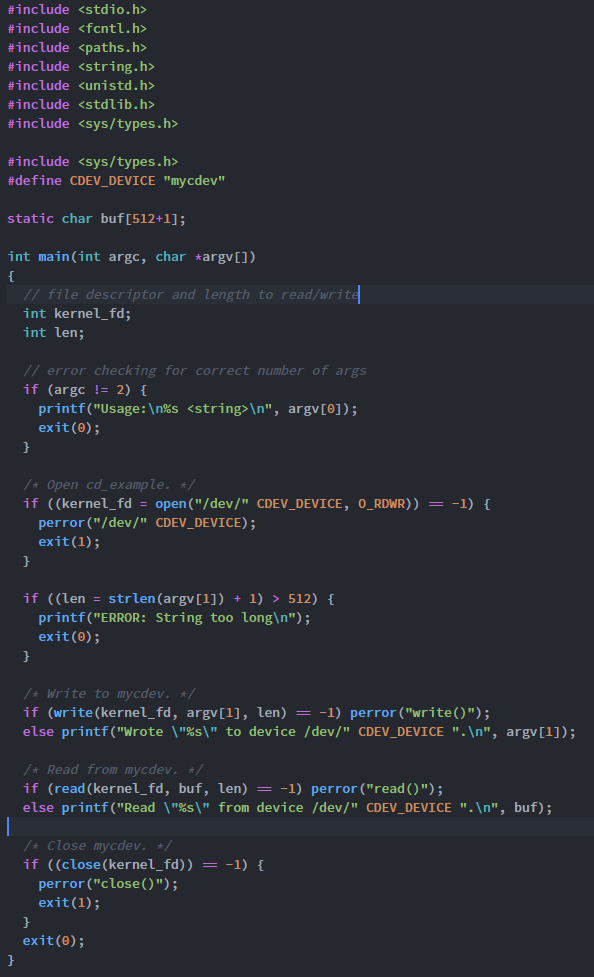
* prot - this defines the protection values for the space. We want to use VM\_PROT\_ALL because this gives it the permissions to read,write and execute from that space (image below)
* max - I am still unsure about this one
* cow - This indicates the flags that should be sent to the new entry. Since we dont have any flags, this is set to 0.



After doing this, we have now successfully allocated space for the new set of execve args.

# How to Test a Character Device

The coding of the file was fairly easy. I followed the steps I had previously outlined [**here**](https://www.openlearning.com/u/roarkmenezes-q6040z/blog/CharacterDeviceModulesKldS/) under "Testing a Character Device". Doing so produced the following code:



To create the executable file I encountered a number of problems. First of all, I was attempting to use the previously used Makefile (that was used for KLD's) to compile the c file. This was giving errors that certain header files couldnt be found. So i tried finding a way around this by re-routing the header files (errors were recieved for stdio.h, stdlib.h, fcntl.h, string.h and paths.h) by prefixing them with ../../inlcude/. This is because the current working directory (cwd) was found to be /sys/. When we run cd .. from that folder we weirdly arrive at usr/src. From here, we need to navigate to usr and then include, hence why I use .. twice. but this still gave numerous errors.

After a few hours of debugging I realised that I wasn't compiling the code as a c program. I was treating it as a KLD. I needed to use FreeBSD's c compiler, cc, to compile it. So after running the code:

**cc -o interface interface.c**

I was then able to arrive at an executable file to give input to my character device.

# Preventing Kernel Data Corruption

An interesting though is what happens when one of the objects that I have hidden is found and killed? If for some reason the kernel decides to do nothing, then all is good. But what if the kernel instead goes and tries to remove the object from its lists and data? The object has already been removed, so it will not be able to delete anything. When the kernel fails to find that data in its lists, it will end up corrupting those data structures in the process, causing all sorts of problems to the kernel and OS. To prevent this we can:

* Hook the terminating function/s e.g. exit etc. to prevent them from removing the objects we have hidden
* Hook the terminating function/s to place the hidden objects back onto the lists before termination.
* Implement my own exit function to safely remove the hidden objects
* Do nothing. If the objects are never going to be found whats the point of implementing safety procedures for their termination?

Since DKOM can only manipulate objects in main memory, it does have limitations, but there are still many objects that can be patched. These can be checked by executing them with the grep -r option in the terminal.

# Hiding a Running Process

Now that we know about the macros and functions associated with how the kernel keeps track of data, and we know about DKOM, we can learn to hide a running process from a user who attempts to debug their computer.

**The proc Structure**

In FreeBSD, the context of each process is maintained in a proc structure, defined in proc.h. There are a few fields in *struct proc* that need to be understood in other to hide a running process:

* LIST\_ENTRY(proc) p\_list - contains *linkage*pointers associated with proc struct. It is referenced during insertion, removal and traversal of the list.
* int p\_flag - process flags, set on a running process. All are defined in proc.h.
* enum {PRS\_NEW = 0, PRS\_NORMAL, PRS\_ZOMBIE} p\_state - represents the current process state. PRS\_NEW = newly born and completely uninitialized process. PRS\_NORMAL = "live" running process, PRS\_ZOMBIE = zombie process.
* pid\_t pid - process ID, 32-bit int value.
* LIST\_ENTRY(proc) p\_hash - not learnt yet
* struct mtx p\_mtx - resource access control. It defines 2 macros, PROC\_LOCK and PROC\_UNLOCK for easily acquiring/releasing this lock.
* struct vmspace \*p\_vmspace - the vm state of the process.
* char p\_comm[MAXCOMLEN + 1] - executes the process. The MAXCOMLEN constant is defined in param.h (19).

**The allproc List**

FreeBSD has 2 lists for its proc structures. All process in ZOMBIE state are in *zombproc*while the rest are in *allproc.*It is referenced by *ps*and *top*commands. Thus you can hide a running process by simple removing its proc structure from the allproc list.

But this got me thinking, wouldnt the process just not run if you delete its proc structure? Since there is no p\_comm entry, the process would not be able to execute. But it turns out the because processes are excuted at *thread granularity,*modifying the process is not complicated.

allproc is defined as: **extern struct proclist allproc.**As this is defined as a proclist structure, and proclist is defined as: **LIST\_HEAD(proclist, proc),**we can see that allproc is just a kernel doubly linked list queue data structure, contained proc structures. The resource access control for the allproc list is defined as: **extern struct sx allproc\_lock.**

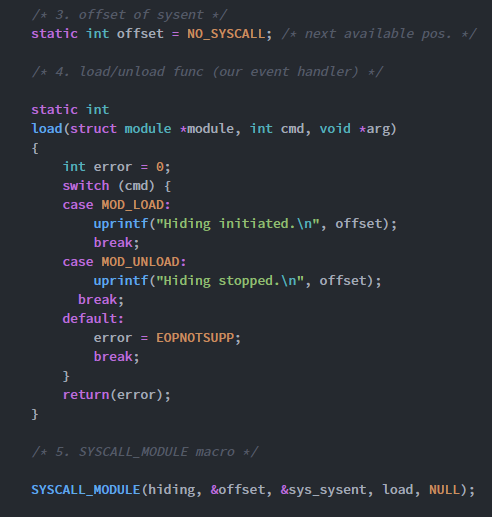
**Trying it out**

To try this out, I used knowledge learnt previous of system call modules and what was just explained, on the doubly linked list macros, locks and proc/allproc structs. So the steps I took to create this system call were:

* Create the system call function, with arguments being the process name, as a char ptr.
  + First we need to acquire the lock on the allproc list
  + Then iterate through the process list, and acquire a lock on each process as we check a number of things:
    - We check to see if the process virtual address space exists (proc->p\_vmspace), if true then unlock and continue
    - We check to see if the process flag is set to working on exiting, if true then unlock and continue
    - We check to see if the proc->p\_comm value is our process name (char ptr), if true, then we remove it from the process list, unlock and continue
  + Once iteration is done, we unlock the allproc resource and return.
* Then we create the sysent structure, offset and event handler functions.
* Finally collate it altogether using the SYSCALL\_MODULE.

Here is the code I wrote to test this out:

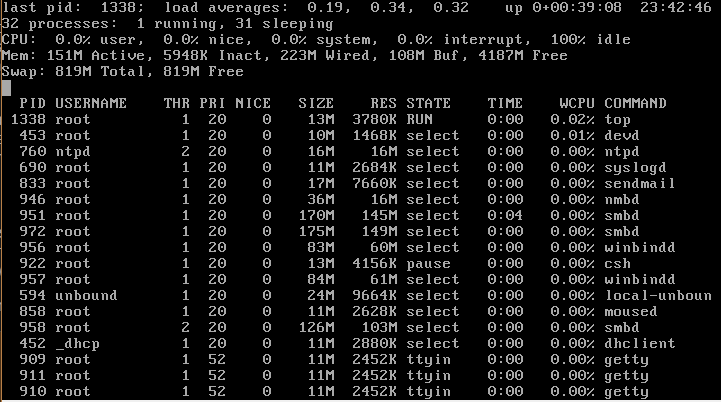




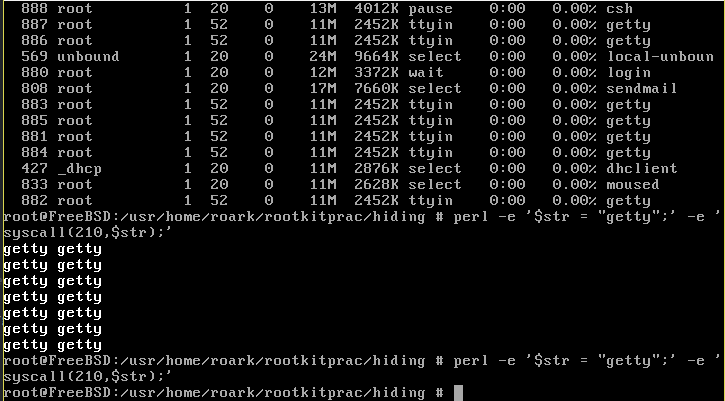
I have inlcuded comments to explain what I am doing. When the code is run in FreeBSD, we can test it out as such, again using perl because its just easier:

I ran the following perl command: **perl -e '$str = "getty"&semi&semi;' -e 'syscall(210, $str)&semi&semi;'**

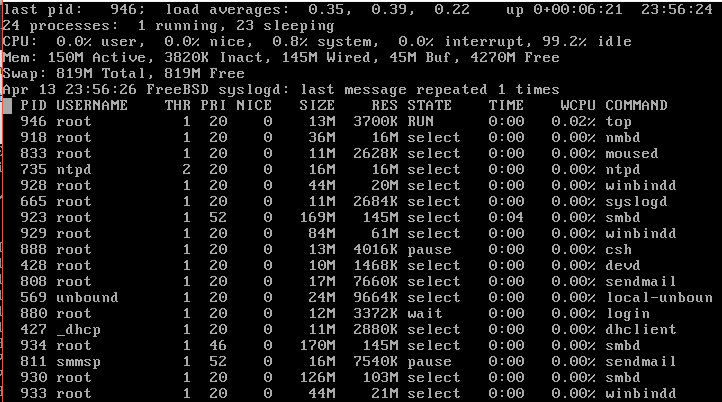
Since the output of top was:



I chose to hide getty. Running the command gives:

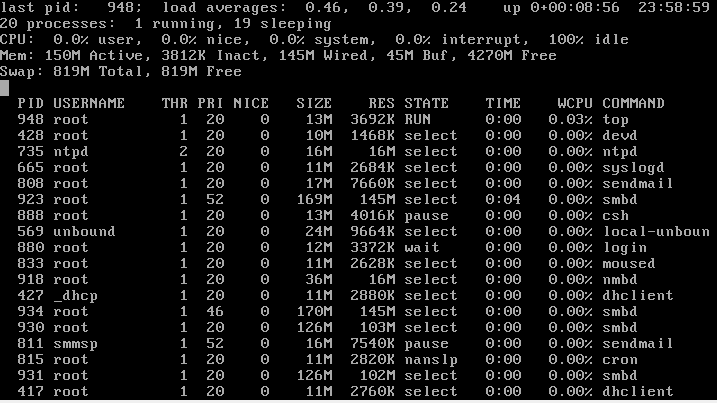


I appended my code to print my string and the process name whenever a match was encountered. So according to this, all 7 getty processes should now be hidden. Running top again shows:



We can see that there are no longer any getty processes. I wanted to try it one more time, with the winbindd process (used for samba). So, i ran the command: **perl -e '$str = "winbindd"&semi&semi;' -e 'syscall(210, $str)&semi&semi;'**

The output of top after running this is:



We can see that the processes are hidden.

From this I have learnt that once I get a trojan process working, I know how to hide it on the victims computer. The rootkit will be doing this part.

# Semi Working Rootkit

# 

# 

Conclusion

Undertaking this project gave me a lot of insight into the world of rootkits, the kernel and security in general. There are so many ways to detect a rootkit, but at the same time those ways become public, hackers find a counter-measure. So its just a never ending race between those in the security industry and hackers. From the knowledge I have gained so far, my next step will be to produce a working rootkit, not on FreeBSD, but on a more commercial OS, such as Windows or Mac. The concepts that I learned are translatable, though obviously not easily, between the FreeBSD OS and other OS's. Overall, this project felt like the beginning for me in my career in security.