CSIE 5374 Assignment 2 (Due on 03/22 23:59)

In assignment 2, you have to write a simple rootkit and provide the following functions: hide/unhide module, masquerade process name, hook/unhook syscall. Rootkit as you might have heard before, is essentially the malware that runs in the kernel space. To achieve these functions, you must implement it as a loadable kernel module (LKM). LKM runs in kernel mode and allows access to all kernel internal structures/functions. It can be used to extend the functionality of the running kernel, and thus it is also often used to implement device drivers to support new hardware.

In this assignment, we provide an LKM template as a starting point for you. You should modify the module source to meet assignment requirements. You should also write a user space program to test the functionality of your rootkit module. Both the rookit and the test program must run on an AArch64 machine. We use QEMU to emulate this, as you did in assignment 1.

In this assignment, you are NOT allowed to modify the kernel source. Your rootkit should work as an independent module on the mainline Linux v5.4.

This is a group assignment, so you should collaborate with your group members on this assignment.

Development Tips

In the following sections, we assume you already know how to compile the linux kernel and create a filesystem image and run it with `qemu-system-aarch64`. Therefore, we will skip those steps already mentioned in assignment 1.

Compile LKM

To compile the kernel module, we assume you have downloaded or installed the following prerequisites.

- Prerequisite
  - Linux v5.4 source code
    - `git clone git@github.com:torvalds/linux.git`
    - `git checkout tags/v5.4`
  - Cross compiler
    - gcc-aarch64-linux-gnu (4:9.3.0-1ubuntu2)
    - `sudo apt update && sudo apt install gcc-aarch64-linux-gnu`
  - LKM template
    - rootkit

We have provided Makefile in the LKM template directory, so you can switch to the LKM template directory and use the `make` command with the following arguments to build the kernel module.

```
$ cd /PATH/TO/rootkit
$ make KDIR=/PATH/TO/linux-5.4-source CROSS=aarch64-linux-gnu-
```

(after `KDIR`, you should specify the path for your kernel source. for example: `KDIR=~/src/linux-5.4/`)

After compilation, A kernel module binary with `.ko` extension is generated.

NOTE 1: You have to make sure the kernel you use in your VM is compiled from the source you specified above.

NOTE 2: Before you test your module, you have to recompile your Linux using the new **defconfig** provided. This defconfig has the config option **CONFIG_KPROBES** for you.

Install LKM

PLEASE DO THIS STEP IN QEMU VM, DO NOT INSTALL IT ON YOUR HOST

After kernel module binary (i.e. `rootkit.ko`) is generated, you can install it to a running kernel using the `insmod` command.

```
$ sudo insmod rootkit.ko
```

You can check installed modules using the `lsmod` command.

```
$ lsmod
```

Once `rootkit` module is installed, you can get major number from `dmesg`

```
$ dmesg | tail
...
[  171.080020] The major number for your device is 236
...
```

then, you can create a character device manually.

```
$ sudo mknod /dev/rootkit c 236 0
```

(change `236` to what you get from `dmesg`)

Finally, you can `open` the file and interact with the module via `ioctl`, `write`...etc.

```
...

#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "rootkit.h"

int main (void) {

  ...

  fd = open("/dev/rootkit", O_RDWR);
```

```
  ...

  ioctl(fd, IOCTL_MOD_HIDE);
}
```

In our template, we only implement `ioctl` file operation. however, you can also define other file operations as you want.

Uninstall LKM

When you want to update a kernel module, you need to remove the old one.

You can use `rmmod` to remove the installed module.

```
$ lsmod
Module                  Size  Used by
...
rootkit                XXXXX  0
...
```

```
$ rmmod rootkit
```

After removing, remember to remove the related device as well.

```
$ rm /dev/rootkit
```

QEMU shared folder (optional)

Frequently updating kernel module binary to filesystem image can be annoying. So it's nice to have a shared folder.

You can follow the instructions below to have a shared folder in QEMU VM.

First, append these two lines to `run-vm.sh`

```
@@ -9,6 +9,7 @@ FS=cloud.img
 CMDLINE="earlycon=pl011,0x09000000"
 DUMPDTB=""
 DTB=""
+SHARED_DIR=./shared

 usage() {
        U=""
@@ -98,3 +99,4 @@ qemu-system-aarch64 -nographic -machine virt,gic-version=2 -m 1024 -
cpu cortex-a
        -append "console=ttyAMA0 root=/dev/vda rw $CMDLINE" \
        -netdev user,id=net0,hostfwd=tcp::2222-:22 \
         -device virtio-net-pci,netdev=net0,mac=de:ad:be:ef:41:49 \
+       -virtfs
local,path=$SHARED_DIR,mount_tag=shared,security_model=passthrough,readonly \
```

Then create a corresponding directory in host.

```
$ mkdir ./shared
```

Finally, boot your QEMU VM up and mount the shared folder somewhere.

```
$ ./run-vm.sh
...

Ubuntu 18.04.6 LTS ubuntu ttyAMA0

ubuntu login: root

...

root@ubuntu:~# mount -t 9p -o trans=virtio shared /mnt
```

Requirements

rootkit (100%)

In this assignment, you should modify the ***ioctl*** system call handler in **rootkit.c** to add new ***ioctl*** numbers for the three different functionality like the following:

```
   switch(ioctl) {
      case IOCTL_MOD_HOOK:
         //do something
         break;
      case IOCTL_MOD_HIDE:
         //do something
         break;
```

```
      case IOCTL_MOD_MASQ:
         //do something
         break;
      default:
        ret = -EINVAL;
    }
   return ret;
```

Hide/Unhide module (10%)

To prevent others from discovering this rootkit through `lsmod`, you must implement a function to remove/add this module from the module list.

The rootkit module should be visible by default. Calling ioctl for the hide functionality hides the module (you will not see it from lsmod); if the module is hidden, making the same ioctl call unhides the module.

HINT 1: you can access module list via `THIS_MODULE->list`

Masquerade process name (30%)

Sometimes it's useful to be able to masquerade the name of a process as another process.

In this requirement, a parameter needs to be passed to the module so that the module knows which process should be renamed and what the new name is. This is what `struct masq_proc` is for.

However, you are asked to handle multiple `struct masq_proc` in a single `ioctl` operation. Hence, you must pass `struct masq_proc_req` to the rootkit module.

The `list` field in `struct masq_proc_req` points to an array of `struct masq_proc`, and the `len` field indicates how many entries are in the `list`.

```
#define MASQ_LEN        16 // <upadted 03/13/22>

struct masq_proc {
        char new_name[MASQ_LEN];
        char orig_name[MASQ_LEN];
};

struct masq_proc_req {
        size_t len;
        struct masq_proc *list;
};
```

In the module, you are required to handle an arbitrary `len`. You should not reserve a fixed-size array to store the contents. You should allocate memory according to the actual data size. You can use the function **kmalloc** to allocate memory, and **kfree** to free the memory allocated by kmalloc.

You should only masquerade process name if the actual length of the ***new_name*** string is shorter than the ***orig_name***. **(either "shorter" or "shorter and equal" are acceptable) <upadted 03/21/22>** You are asked to iterate the list of ***struct masq_proc***, and try to masquerade process name using the data from each of the entries if possible.

HINT 1: If the function succeeds, you should observe the results from ***ps ao pid,comm***. **<upadted 03/13/22**>

Hook/Unhook syscall (50%)

Hooking syscall means to alter the behavior of syscall by intercepting it. You can achieve this by overwriting syscall table entries.

In this assignment, you have to hook the following syscalls and perform corresponding operations.

1. **execve** (20%)

   record all paths executed by `execve` to `dmesg`.

   ```
   $ dmesg
   ...
   [ 3946.454859] exec /bin/ls
   [ 3952.944417] exec /bin/dmesg
   ...
   ```

2. **reboot** (20%)

   your system call hook should intercept the request to power off, and forbid it from happening.

   ```
   $ poweroff
   ...
   [  OK  ] Stopped target Swap.
   [  OK  ] Reached target Shutdown.
   [  OK  ] Reached target Final Step.
            Starting Power-Off...
   //stop here
   ```

3. **bonus** (10%)

   You are asked to hook another system call in Linux to perform any other cool hacks.

   For this part, you should submit a write-up about how to trigger the hack. Preferably, you should submit a test program for your new hook.

Hint 1: In the AArch64 kernel, syscall table (sys_call_table) is defined in **arch/arm64/kernel/sys.c** as a constant. It is located in the `.rodata` segment in the compiled kernel binary, which is a read-only segment. To circumvent this, you may use `update_mapping_prot` to update the Write permission of the memory segment.

Hint 2: Since your rootkit is compiled as a kernel module, it does not have access to some kernel symbols. Linux only exports a certain sets of symbols to module developers. For missing symbols that you with to you, you could then use kprobe to search for the function **kallsyms_lookup_name**, which will be useful to find other missing kernel symbols.

Life is cruel, so the recent kernel versions actually do not export **kallsyms_lookup_name** to you. To get access to it from your module, you can append the following to your module code:

```
#include <linux/kprobes.h>
static struct kprobe kp = {
  .symbol_name = "kallsyms_lookup_name"
};

typedef unsigned long (*kallsyms_lookup_name_t)(const char *name);
      unsigned long start_rodata, init_begin, section_size = 0;
      kallsyms_lookup_name_t kallsyms_lookup_name;

      register_kprobe(&kp);
      kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr;
      unregister_kprobe(&kp);

//char str[10] = "hello";
//your_function_ptr = (void *)kallsyms_lookup_name(str);
```

Hint 3: The symbols (functions or global variables) defined in your module will go away after you remove the loaded module. The kernel will not have access the symbols.

Hint 4: As we discussed in the lecture, system call parameters are passed via general purpose registers in Arm (e.g. x0, x1, x2). You might want to figure out how system call parameters were passed by the kernel originally.

Write-up (10%)

Each of the groups are required to provide a write-up about the assignment. The write-up should include some explanation of your source code (ex: how it works), description for how you test the rootkit, and contribution from each of your group members.

Homework submission

You should submit the assignment via NTU Cool.

Submission Format

For the rootkit, you will be required to submit source code, a Makefile, and a write-up file. Compress all the files in hw2.zip, and upload the zip file to NTU Cool.

Your Makefile does not have to deal with copying or installing the rootkit in your VM. We recommend you use the default Makefile that we provide.

```
.
├── source code of your user space test program
├── Makefile
├── write-up.md
├── rootkit.c
├── rootkit.h
├── any extra file
```

Each of the groups only need to submit one copy of the assignment.

Grading criteria

You will get zero points if your code fails to compile. Please run checkpatch against the source code for your rootkit module. You will lose points if checkpatch reports either warnings or errors.

You are required to properly manage resources (free allocated memory), handle errors, and return error codes to user space.

Plagiarism policy

You are allowed to reference sources from the internet. If you do, please attach all of your references in the write-up. If we find that your code is similar to other's from the internet, we will count it as plagiarism if we could not find the corresponding references. You will automatically get zero points for the assignment.

Late Policy

We do not accept late submissions for this assignment. Please start the assignment early.