CMSC 421: Principles of Operating Systems

Project 1: SHA-3 Kernel Module

This project is due on Tuesday, November 20, at 11:59:59 PM (Eastern standard time). You *must* use the *submit* to turn in your homework like so: `submit cs421_jtang proj1 sha3.c sha3-test.c`

Your module code must be named `sha3.c`, and it will be compiled against a 4.17 Linux kernel source tree, via the Kbuild supplied below. It must not have any compilation warnings; **warnings will result in grading penalties**. This module code must be properly indented and have a file header comment, as described on the coding conventions page. Prior to submission, use the kernel's indentation script to reformat your code, like so:

```
~/linux/scripts/Lindent sha3.c
```

In addition, you will write a unit test program, `sha3-test.c`, and it will be compiled on Ubuntu 16.04 as follows:

```
gcc --std=c99 -Wall -O2 -pthread -o sha3-test sha3-test.c -lm
```

There must not be any compilation warnings in your submission; **warnings will result in grading penalties**. In addition, this code must also have a file header comment and be properly indented. You will submit this test code along with your module code.

The Secure Hashing Algorithm (SHA-1) is one of the fundamental cryptographic algorithms underlying many modern-day Internet security. SHA-1 is used to authenticate websites and documents; without it attackers can easily impersonate people and institutions (such as your bank website). In February 2017, researchers at Google announced a practical attack on SHA-1. Thus it is prudent for people to switch to more secure algorithms.

A *hashing algorithm* takes some arbitrary input data and calculates a unique identifier. This unique identifier is known as the *message digest* (or simply digest) for the input. Hashing algorithms form the basis of modern cryptography, from digital code signing, secure banking, and for mining Bitcoins. Any change to the input, regardless of how minor it is, will result in a completely different digest. Furthermore, given a particular digest, it is infeasible for a person to derive the original input data, assuming the hashing algorithm is secure. As noted in the Google blog post above, SHA-1 is now known to fail this property.

As an example of how a hashing algorithm works, the input string `CMSC 421` will result in the following SHA-3 512-bit digest:

```
ae1cdc700fee269bece4e58a629b4666c5cd10acb85ced551634b53f4f5da51d37a4609bd1e291d349ffc1e592fb3667bf62c1e267315bec8bac119c0a6ab9dc
```

Try out other input strings via this handy online SHA-3 implementation.

In this project, you will write a Linux kernel module that allows users to hash arbitrary messages via the relatively new SHA-3 algorithm. Your module will create two miscellaneous devices, `/dev/sha3_data` and `/dev/sha3_ctl`. The user sets the data to hash by creating a memory map, via `mmap()`, to `/dev/sha3_data` and then writing to that map; alternatively the user can write directly to the device. The user then writes to `/dev/sha3_ctl` the number of bytes to hash; a subsequent read from `/dev/sha3_ctl` returns the message digest. To simplify this assignment, you will not implement the SHA-3 algorithm itself; instead, you will reuse Linux's implementation.

Part 1: Compile sha3 Module

> ***All instructions henceforth assume you successfully completed the first homework. If you have not done so, go back and finish the homework before proceeding. You have been warned.***

To begin, create a directory for your project and download the following files into that directory via `wget`:

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj1/sha3.c
: Skeleton code for your SHA-3 kernel module.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj1/sha3-test.c
: Skeleton code for your unit test code.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj1/Kbuild
: Read by Linux kernel's build system, defines what is being built. You do not need to modify this file, nor should you submit it with your work.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj1/Makefile
: Builds the kernel module and unit test program, by simply running `make`. Also included is a *clean* target to remove all built objects. You do not need to modify this file, nor should you submit it with your work.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj1/0001-x86-Update-cs421_defconfig-to-build-sha3-kmod.patch
: Updates the kernel `cs421_defconfig` that was set in the first homework.

The first thing to do is to update your kernel configuration. Copy the patch file above into your Linux kernel source tree. Apply the patch via `git am`; see part 4 of the first homework. Then run `make cs421_defconfig` to recreate the configuration. This will enable the `sha3_generic` kernel module. Recompile the kernel, then reinstall all kernel modules. (It is not necessary to reboot the system.)

Next, return to the directory containing your downloaded files. Run `make` to compile everything. You will get some warnings about unused symbols; by the end of this project you will have used all of them. You should now have the kernel module `sha3.ko`. Load that module like so:

```
sudo insmod sha3.ko
```

The module was inserted if the following returns your `sha3`:

```
lsmod | grep sha3
```

So far, all this module does is write a message to the kernel's ring buffer. View the module messages like so:

```
dmesg | tail
```

The number at the beginning is the time stamp of when the message was written. To unload your module, run this command:

```
sudo rmmod sha3
```

Re-examine the ring buffer to see the message generated during module exit. Every time you make a change and recompile `sha3.c`, you will need to first unload the module and then reinsert it.

When the `sha3` module is loaded, the kernel calls its *init* function (similar to a C program's `main()` function), where execution begins. Currently, this module's *init* function calls `pr_info()` and allocates some memory for itself. The `pr_info()` function is an easy way to generate logging messages within kernel code. It accepts a format string like `printf()`, but has additional format specifiers useful for kernel programming.

Part 2: Create Data Miscellaneous Device

You are about to make changes to your Linux kernel. There is a slight chance of accidentally erasing your virtual machine's hard drive. Create a snapshot of your virtual machine before proceeding.

Creating a custom character device can be daunting, and in this project, you will create *two* of them. Fortunately, the Linux kernel has the miscellaneous devices subsystem to simplify this task. For this project, the `sha3` module will use *miscdevice* to control `/dev/sha3_data` and `/dev/sha3_ctl`.

Start off by examining `sha3.c`, specifically the stub functions `sha3_data_read()` and `sha3_data_write()`. Follow these steps to create the device `/dev/sha3_data`:

1. Create a global variable of type `static const struct file_operations` to handle `/dev/sha3_data`. Set its `read` callback to `sha3_data_read`, `write` callback to `sha3_data_write`, and `mmap` callback to `sha3_data_mmap`.
2. Create a global variable of type `static struct miscdevice` for `/dev/sha3_data`. Set its `minor` field to `MISC_DYNAMIC_MINOR`, `name` field to `"sha3_data"`, `fops` field to point to the previously created `struct file_operations`, and `mode` callback to `0666`.
3. In `sha3_init()`, call `misc_register()` to create the character device. In `sha3_exit()` call `misc_deregister()` to undo the registration.

If all of the above works, when the module is loaded, you will now have a character device `/dev/sha3_data`:

```
$ sudo insmod sha3.ko
$ ls -l /dev/sha3*
crw-rw-rw- 1 root root 10, 56 Jan  1 12:00 /dev/sha3_data
$ echo -n 'CMSC 421' > /dev/sha3_data
bash: /dev/sha3_data: Operation not permitted
```

If implemented incorrectly, your kernel ring buffer may contain a message that looks like this:

```
[32409.452666] do_init_module: 'sha3'->init suspiciously returned 1, it should follow 0/-E convention
               do_init_module: loading module anyway...
[32409.452669] CPU: 0 PID: 26980 Comm: insmod Tainted: G           O    4.9.5+ #3
[32409.452670] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[32409.452673]  ffff9852c0333d28 ffffffffae2b4e4b ffffffffc026c000 0000000000000001
[32409.452684]  ffff9852c0333d50 ffffffffae10b30d ffff9852c0333eb0 0000000000000001
[32409.452686]  ffffffffc026c000 ffff9852c0333e90 ffffffffae0bfc82 ffffffffc026c000
[32409.452688] Call Trace:
[32409.452702]  [<ffffffffae2b4e4b>] dump_stack+0x4d/0x72
[32409.452706]  [<ffffffffae10b30d>] do_init_module+0x83/0x1e6
[32409.452709]  [<ffffffffae0bfc82>] load_module+0x22e2/0x2760
[32409.452713]  [<ffffffffae0bcca0>] ? __symbol_put+0x30/0x30
[32409.452725]  [<ffffffffae0c030c>] SYSC_finit_module+0xbc/0xf0
[32409.452727]  [<ffffffffae0c0359>] SyS_finit_module+0x9/0x10
[32409.452732]  [<ffffffffae52f824>] entry_SYSCALL_64_fastpath+0x17/0x98
```

Read the error message to determine where within your code caused the fault that the Linux kernel had detected. Most errors are unrecoverable; a symptom of an unrecoverable condition is when the kernel refuses to unload the driver. In this case, your only recourse is to reboot the virtual machine.

Because `sha3_data_write()` returns `-EPERM`, trying to write to the device is supposed to result in permission denied. Observe the `-n` flag passed to `echo`; this prevents `echo` from automatically adding a trailing newline to the output. Also note that `echo` never writes the trailing `\0` character. In other words, the above command attempts to write exactly 8 bytes to `/dev/sha3_data`, not 9 nor 10.

The next step is to implement `sha3_data_read()` and `sha3_data_write()` as per their comments. For these functions, the module cannot simply access the `ubuf` pointer. Instead, the code **must** use `copy_to_user()` and `copy_from_user()` to safely copy data to and from user space into kernel space.

After implementing these callbacks, recompile and reinsert the module. Test it like so:

```
$ echo -n 'CMSC 421' > /dev/sha3_data
$ xxd /dev/sha3_data | head
00000000: 434d 5343 2034 3231 0000 0000 0000 0000  CMSC 421........
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Recall that the handy `xxd` utility was used in the second homework to display human-readable output from a program. Here, it is used to display the raw bytes from a given file in hexadecimal format. Because this assignment deals with individual bytes, not strings nor other ASCII visible characters, you will need to get used to reading byte values as expressed in hexadecimal.

Part 3: Create Control Miscellaneous Device

Now that you can read and write to the data buffer, the next step is to hash the data buffer contents. Examine the stub functions `sha3_ctl_read()` and `sha3_ctl_write()`. Follow these steps to create the device `/dev/sha3_data`:

1. Create a global character array to hold the message digest. Experiment with `sha3_digest()` to determine exactly how many bytes to allocate for this array.
2. Create a global variable of type `static const struct file_operations` to handle `/dev/sha3_ctl`. Set its `read` callback to `sha3_ctl_read` and `write` callback to `sha3_ctl_write`.
3. Create a global variable of type `static struct miscdevice` for `/dev/sha3_ctl`. Set its `minor` field to `MISC_DYNAMIC_MINOR`, `name` field to `"sha3_ctl"`, `fops` field to point to the previously created `struct file_operations`, and `mode` callback to `0666`.
4. In `sha3_init()`, call `misc_register()` to create the character device. In `sha3_exit()` call `misc_deregister()` to undo the registration.
5. Implement `sha3_ctl_read()` as per its comments. `sha3_ctl_read()` will need to properly call `copy_to_user()`.
6. Implement `sha3_ctl_write()` as per its comments. `sha3_ctl_write()` instead should use `kstrtoul_from_user()` to determine how many bytes to hash. This function then calls the already provided `sha3_digest()` to hash the contents of `sha3_data_buffer` into your global digest array. **Note the final parameter to `sha3_digest()` is an in/out variable.**

After implementing all four callbacks, recompile and reinsert the module. Test it like so:

```
$ echo -n 'CMSC 421' > /dev/sha3_data
$ echo -n '8' > /dev/sha3_ctl
$ xxd /dev/sha3_ctl
00000000: ae1c dc70 0fee 269b ece4 e58a 629b 4666  ...p..&.....b.Ff
00000010: c5cd 10ac b85c ed55 1634 b53f 4f5d a51d  .....\.U.4.?O]..
00000020: 37a4 609b d1e2 91d3 49ff c1e5 92fb 3667  7.`.....I.....6g
00000030: bf62 c1e2 6731 5bec 8bac 119c 0a6a b9dc  .b..g1[......j..
```

Note how this digest matches the one at the top of this page.

Part 4: Develop Unit Tests and Add Documentation

Now that you have (in theory) a working module, you must then write your own unit tests. Modify `sha3-test.c` to open `/dev/sha3_data` and `/dev/sha3_ctl`. Create a memory mapping to `/dev/sha3_data`. This program is to exercise all of the functionality as described above. This includes a mix of inputs when writing to the device nodes, reading from and writing to the memory map, and verifying that the generated digests read from `/dev/sha3_ctl` are correct.

When calling `mmap()`, use `PAGE_SIZE` as the number of bytes to map. `PAGE_SIZE` is defined in the header `<sys/user.h>`.

You will need to create your own testing framework; as a suggestion, reuse the one employed in homework 4. The unit tests must have comments that explain what things are being tested. Your goal is to test boundary conditions of your miscellaneous devices; you will be graded based upon the thoroughness of the tests. For example, you are responsible for checking that the user cannot hash more than one `PAGE_SIZE` worth of data.

As that your tests will perform multiple reads and writes from the devices, you will probably need to reposition your file pointer after each operation. Also keep in mind that digests are unsigned bytes; thus store digests in variables of type `unsigned char`, not `char` (which are signed on x86-64).

Other Hints and Notes

- Ask plenty of questions on the Blackboard discussion board.
- At the top of your submission, list any help you received as well as web pages you consulted. Please do not use any URL shorteners, such as goo.gl or TinyURL. Also, do not cite shared data services, such as Pastebin, Dropbox, or Google Drive.
- Use the Linux Cross-Reference website to quickly search through kernel source code.
- You may modify any of the provided code. You may need to add more functions and global variables than those listed above.
- The input data buffer holds up to one `PAGE_SIZE` amount of characters. This does not mean `PAGE_SIZE` - 1 + trailing `\0`, but exactly one `PAGE_SIZE` worth of bytes.
- Make sure you indent your code one last time prior to submission. **Unindented kernel code will result in grading penalties.**

Extra Credit

Sorry, there is no extra credit available for this assignment.