CMSC 421: Principles of Operating Systems

Project 2: Password Keeper

This project is due on Sunday, December 16, at 11:59:59 PM (Eastern standard time). You *must* use the *submit* to turn in your assignment like so: `submit cs421_jtang proj2 pwkeeper.c pwkeeper-test.c`

Your driver code must be named `pwkeeper.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 pwkeeper.c
```

In addition, you will write a unit test program, `pwkeeper-test.c`, and it will be compiled on Ubuntu 18.04 as follow:

```
gcc --std=c99 -Wall -O2 -pthread -o pwkeeper-test pwkeeper-test.c cs421net.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 driver code.

Within just the past few months many Internet sites have experienced highly visible data breaches affecting millions of customers. The best recourse, other than to live without the Internet, is to use different passwords for every website. Doing so is impractical; a 2016 survey found that the average person has 27 different online accounts. One solution is a password manager, a tool that stores and generates passwords. In this project, you will create your own password manager as a Linux driver. If implemented correctly, your password system may be more secure than the professionals!

Part 1: Obtain Necessary Files

All instructions henceforth assume you successfully completed the first project. If you have not done so, go back and finish that assignment 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/proj2/pwkeeper.c
    Skeleton code for your kernel driver.
http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/pwkeeper-test.c

Skeleton code for your unit test code.

In addition, download these files into that same directory. You will not need to modify any of these files, nor should you submit any of them with your work.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/Kbuild
: Read by Linux kernel's build system, defines what is being built.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/Makefile
: Builds the kernel module and unit test program, by simply running `make`. Also included is a *clean* target to remove all built objects.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/0001-x86-irq-Raise-interrupt-for-registered-CPU.patch
: Updates the simulated "CS421Net" network device, used to generate interrupts.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/xt_cs421net.c
: A *Netfilter module* that simulates a network device.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/xt_cs421net.h
: Header file that declares symbols defined in `xt_cs421net.h`.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/cs421net.c
: Adds networking functions to your unit test code.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/cs421net.h
: Header file that declares symbols defined in `cs421net.c`.

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/proj2_start.sh
: Shell script that reconfigures network settings needed for this project. After downloading this file, mark the script executable (`chmod u+x proj2_start.sh`).

http://www.csee.umbc.edu/~jtang/cs421.f18/homework/proj2/proj2_stop.sh
: Shell script that disables network settings that were needed for this project. After downloading this file, mark the script executable (`chmod u+x proj2_stop.sh`).

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. Recompile and reinstall the kernel, then reboot to make the changes go into affect.

Now run `make` to compile everything. Upon success, you should now have the kernel driver `pwkeeper.ko` and user space program `pwkeeper-test`. Run `proj2_start.sh`; it may ask for your password, as that it runs some commands under `sudo`. This will set up your VM for this project. Rerun this script if you ever need to reboot your VM.

Part 2: Store Users' Master Passwords

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.

The first task is to allow users to register their own master passwords within the system. In `pwkeeper.c`, read the comments for `pwkeeper_master_write()` and `pwkeeper_remove()`. At the top of the file, declare a kernel linked list. Each element of the list holds two entries: a user ID and that user's master password. Because this master password list can be accessed by multiple threads, guard accesses to the list using a spin lock, specifically using the `irqsave` form.

Next, modify `pwkeeper_master_write()`. If the current user has already set a master password, replace it with the incoming value. Otherwise, allocate a new list entry for that user and set his or her password. Add that entry to the master password list with the `list_add_tail()` macro. Then in `pwkeeper_remove()`, free all memory that was allocated for this master password list.

Master passwords are limited to 32 bytes. They are not limited to printable ASCII characters, nor are they necessarily C-style strings. If the user does not specify all 32 bytes, or replaces the password with one that is shorter, pad the remaining bytes with the null character.

Your kernel driver will need to obtain the user ID (UID) of the requesting process. Be aware of various blog posts describing the `current_uid()` call. Kernel UIDs have changed; many older blog posts are inaccurate. In modern kernels, UIDs are no longer scalars but rather are `struct` types. See the kernel code itself.

Test that your code correctly allows different users to set their own master passwords. Add additional users to your VM. Then use `su` to log in as another user.

Part 3: Generate Account Passwords for Users

The next task is to allow users to add accounts and auto-generate passwords for those accounts. In `pwkeeper.c`, read the comments for `pwkeeper_account_read()`, `pwkeeper_account_write()` and `pwkeeper_remove()`. At the top of the file, declare another kernel linked list. Each element of this second list holds three entries: a user ID, an account name, and then the password for that account. Because this accounts list can be accessed by multiple threads, guard accesses to this list using a spin lock, specifically using the `irqsave` form.

Next, modify `pwkeeper_account_write()`. If the current user has not created the account, allocate a new list entry containing the user ID, account name, and soon-to-be generated password. Add that entry to the accounts linked list. If the given account was already added, do nothing. Then in `pwkeeper_remove()`, free all memory that was allocated for this accounts list.

Account names are limited to 16 bytes. They are not limited to printable ASCII characters, nor are they necessarily C-style strings. If the user does not specify all 16 bytes, pad the remaining bytes with the null character.

When an account is added, your driver will automatically generate a password for that account. Implement the following key derivation function to generate the password:

1. Declare a temporary buffer that is 48 bytes in length.
2. Set the first 32 bytes of the buffer to the user's master password.
3. Set the last 16 bytes to the account name.
4. Hash the buffer using the SHA-3 algorithm to derive a 64-byte digest.
5. Take the lower 6 bits of the first byte of the digest to get a number between 0 and 63. Add 48 to that number. Store the resulting number as the first byte of the password.
6. Repeat the above step with the next 15 bytes of the digest to obtain a 16 character password. Store this password in the accounts linked list.

When a user resets his or her master password, perform this KDF again for all accounts associated with that user.

Now that your driver can create account passwords, implement the code for `pwkeeper_account_read()`. Return to the user the password associated with the most recently written account name.

If all of the above works, compile and insert your module. Test it like so:

```
$ echo -n '01234567890123456789012345678901' > /dev/pwkeeper_master
$ echo -n 'abcdEFGHijklMNOP' > /dev/pwkeeper_account
$ cat /dev/pwkeeper_account && echo
UMWd8^TH\]RkTXLR
```

Part 4: Add Accounts Viewing

The next feature to add is account viewing. You will implement two sysfs callbacks to show the state of your Password Keeper system. The first one shows all of accounts and passwords registered to the current user. The second callback shows all users who have registered master passwords, but only if the current user is privileged.

Read the comments for `pwkeeper_accounts_show()` and `pwkeeper_users_show()`. When the `pwkeeper_accounts_show()` callback is invoked, iterate across the list of account passwords. For each entry matching the current UID, write to the buffer the account name and password. Write the entire account name, even if contains non-printable and/or whitespace characters.

For `pwkeeper_master_show()`, check if the calling process has the `CAP_SYS_ADMIN` capability. If it does not, reject the request. Otherwise, iterate across the master password list, writing to the buffer the list of all UIDs. Use the `%u` format specifier when writing a UID to the output buffer.

If all of the above works, compile and insert your module. Supposing that your UID is 1000, you should get the following output after recompiling your driver:

```
$ sudo insmod pwkeeper.ko
$ echo -n 'cs421_rocks!' > /dev/pwkeeper_master
$ echo -n 'umbc.edu' > /dev/pwkeeper_account
$ echo -n 'gmail.com' > /dev/pwkeeper_account
$ echo -n 'snapchat.com' > /dev/pwkeeper_account
$ cat /sys/devices/platform/pwkeeper/accounts
Account  Password
-------  --------
snapchat.com  o>Phj`eI6CX5NOgJ
gmail.com  Y]o8`F\XCeBW1AY\
umbc.edu  kVcXI8`RP086K5bD
$ cat /sys/devices/platform/pwkeeper/masters
cat: /sys/devices/platform/pwkeeper/master: Operation not permitted
$ sudo cat /sys/devices/platform/pwkeeper/masters
Registered UIDs
---------------
1000
```

Part 5: Add Interrupt Handling

Just as smartphones can be erased remotely, your Password Keeper system can also be controlled remotely. Read the code in the file `xt_cs421net.c`. When this Netfilter module is installed, via the `proj2_start.sh` script, it will raise an interrupt every time your computer receives network packets on TCP port 4210. It is your driver's responsibility to handle those interrupts, by deleting the requested UID.

In `pwkeeper_probe()`, install a threaded interrupt handler for interrupt number `CS421NET_IRQ`. Remove that handler in `pwkeeper_remove()`. Then implement `cs421net_top()` and `cs421net_bottom()`. The user will send a single TCP packet with a payload containing a 32-bit little-endian unsigned integer. *The payload is **not a string**; it is not NULL-terminated.* Then iterate across the account and master lists, deleting all nodes whose UID matches the payload. *Ensure your code does not cause any memory leaks!*

As that `cs421net_bottom()` will be accessing your account linked list, be sure to guard the code with a spin lock. (Because the bottom half is not running in interrupt context, you do not need to use `spin_lock_irqsave()`).

Once you are confident your ISR works, call `cs421net_enable()` in `pwkeeper_probe()`, and likewise disable network integration via `cs421net_disable()` in `pwkeeper_remove()`. Install your module, and check `/proc/interrupts` to ensure your ISR was registered.

To test this part, ensure you earlier ran `proj2_start.sh`. Then use `echo` to pipe raw bytes into the handy `nc` tool, which then sends those bytes to port 4210. Again, supposing that your UID is 1000 (hexadecimal 0x000003e8), you should get the following output after recompiling your driver:

```
$ sudo insmod pwkeeper.ko
$ echo -n 'My Proj2 Part 5' > /dev/pwkeeper_master
$ echo -n 'instagram.com' > /dev/pwkeeper_account
$ cat /sys/devices/platform/pwkeeper/accounts
Account  Password
-------  --------
instagram.com  6HQ5ZiG[E781:_?9
$ echo -n -e '\xe8\x03\x00\x00' | nc -N localhost 4210
$ cat /sys/devices/platform/pwkeeper/accounts
Account  Password
-------  --------
$ sudo cat /sys/devices/platform/pwkeeper/masters
Registered UIDs
---------------
```

Part 6: Testing and Documentation

Now that you have (in theory) a working driver, you must then write your own unit tests. Read the networking code in `cs421net.h`. Use `cs421net_send()` to simulate sending network bytes to your Password Keeper system . Modify the `pwkeeper-test.c` to exercise all functionality of this assignment. This includes a mix of inputs when writing to the device nodes and network socket, confirming that users can set only their own master password, that users can add account names, and verifying network login attempts. Two functions that may be of use are `getuid()` and `getresuid()`. You should avoid using `system()`, and instead learn how to perform I/O yourself.

The unit tests must have comments that explain what things are being tested. As before, your goal is to test boundary conditions of your driver's interfaces. You will be graded based upon the thoroughness of the tests.

Assume that grader will run your unit tests from a "fresh" installation. That is, the grader will not have written anything to `/dev/pwkeeper_master` nor `/dev/pwkeeper_account` prior to running your unit tests. If your unit test program requires to be run as the root user (i.e., under `sudo`), indicate as such within a comment at the top of the file.

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. Although the Linux kernel does not use the standard C library, it does reimplement common glibc functions.
- Use the `KUIDT_INIT` macro to create a `kuid_t` from a scalar.
- You may modify any of the provided code. You may need to add more functions and global variables than those listed above.
- Make sure all resources are released in all error paths. This includes unlocking spin locks, freeing kernel memory, freeing the IRQ, and deregistering miscellaneous devices.
- Make sure you indent your code one last time prior to submission. **Unindented kernel code will result in grading penalties.**

Extra Credit

You may earn an additional 15% credit for this assignment by improving your KDF. The astute observer will note that if by chance two different users happen to select the same master passwords, then the generated passwords will be the same if they also chose the same account names. The remedy for this situation is to salt the KDF. There are many ways to salt passwords; the following is relatively cryptographically secure.

In your accounts linked list declaration, add a salt field to each accounts entry. This field is a 16 byte array. In `pwkeeper_account_write()`, fill the salt array's contents randomly, by using `get_random_bytes()`. Then as part of the KDF, increase the temporary buffer size from 48 to 64 bytes. Prepend the salt to the buffer, such that the buffer's first 16 bytes are the salt, next 32 bytes are the master password, and final 16 bytes are the account name. Do not display the salt when the user reads from `/sys/devices/platform/pwkeeper/accounts`.

Afterwards, update `pwkeeper-test.c` to test this new functionality. Explicitly add a unit test that demonstrates that the same combination of master password and account name will result in different passwords [with extremely high probability]. Then add another unit test that shows if two different users have the same master password and account name, their generated passwords will be different [with extremely high probability].

If you choose to perform this extra credit, put a comment at the top of your file, alerting the grader.