1. Setup cgroup controls to be passed into the program using command line arguments.

The file **sr_container.c** has an array **'*cgroups*'** which is supposed to hold all the cgroup controls for the newly created container. This array holds structs of type "**cgroups_control**". You would ideally have one entry of this struct inside the **'cgroups'** array per cgroup control (memory, cpu, cpuset, blkio). Within this struct you have a double pointer **'settings'** that points to a collection of struct type **'cgroup_setting**'. This struct holds settings specific to a cgroup-controller.

Ex: cgroup-controller: memory
cgroup-settings: memory.limit_in_bytes,
memory.kmem.limit_in_bytes
tasks

So you must fill in the **cgroups** array with **'cgroups_control'** elements. And each of these elements will have a list of relevant settings as shown above. The given code has an example for the **'blkio'** controller. Note that all controls will have the **'tasks'** setting to ensure the process is added to the tasks list of that cgroup.

You must update the **main()** of the given program to support more flags. These flags will enable the user (of the program) to set cgroups when running the code. You must accordingly fill in the above array with the right values. You can have a look at how the arguments are handled in as of now in main() and extend it to fetch more flags and update the array. ***The flags to be supported are given as comments in the template code.*** **Note that the 4th flag was changed from blkio-weight to memory.**

In addition to the cgroup controls, an addition flag also is to be supported to provide the program with a ***hostname***. The value of this flag must be set to the **'*hostname*'** attribute of **'*child_config*'** struct created at the beginning of main().

2. Implement the child process creation logic

Fill in the left off portion of the code in **main()** [in **sr_container.c**] to successfully create a child process with namespace isolation for the following namespace: **`Network, Cgroup, PID, IPC, Mount, UTS`** **(Don't add User namespace) . Lines 171 - 186.**

3. Changing root using pivot_root()

Complete the method **switch_child_root()** in the **sr_container_helpers.c** file using the pivot_root() system call. Refer here for info on the arguments to use with pivot_root(): http://man7.org/linux/man-pages/man2/pivot_root.2.html

4. Setting capabilities to the container

For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: *privileged* processes (whose effective user ID is 0, referred to as superuser or root), and *unprivileged* processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

In recent kernel versions, Linux divides the privileges traditionally associated with superuser into distinct units, known as *capabilities*, which can be independently enabled and disabled. Thus with this new feature the kernel can control privileges allowed to a traditional super-user process.

Capabilities basically subdivide the the property of being “root”. We can restrict certain access of some processes even though they have root privileges. For example we may allow a process to set network devices (**CAP_NET_ADMIN**) but disallow reading all files (**CAP_DAC_OVERRIDE**). However, not all of the properties of being a root is subdivided into capabilities. There are some properties that is still accessible after dropping capabilities.
Read here for more info: http://man7.org/linux/man-pages/man7/capabilities.7.html
(You can complete the assignment even with the description on this handout)

In this assignment we want some of these harmful/unnecessary capabilities also to be disabled from our SRContainer. The list of capabilities that must be disabled are:

CAP_AUDIT_CONTROL, CAP_AUDIT_READ, CAP_AUDIT_WRITE, CAP_BLOCK_SUSPEND, CAP_DAC_READ_SEARCH, CAP_FSETID, CAP_IPC_LOCK, CAP_MAC_ADMIN, CAP_MAC_OVERRIDE, CAP_MKNOD, CAP_SETFCAP, CAP_SYSLOG, CAP_SYS_ADMIN, CAP_SYS_BOOT, CAP_SYS_MODULE, CAP_SYS_NICE, CAP_SYS_RAWIO, CAP_SYS_RESOURCE, CAP_SYS_TIME, CAP_WAKE_ALARM

Disabling capabilities involves 2 steps:

Dropping the said capability from the **ambient capability set** of the process.
Clearing the said capability from the **inheritable capability set** of the process.

You can read more about the different capability sets of a process in the man page. However the description that follows must be sufficient to complete the assignment.

Ex: Say you want to disable the capabilities: ***CAP_MKNOD*** and ***CAP_SYS_BOOT***

```c
int drop_caps[] = { CAP_MKNOD, CAP_SYS_BOOT};
size_t num_caps_to_drop = 2;

// STEP (1) :: Dropping the capability from the AMBIENT set
for (size_t i = 0; i < num_caps_to_drop; i++)
{
    if (prctl(PR_CAPBSET_DROP, drop_caps[i], 0, 0, 0))
    {
        fprintf(stderr, "prctl failed: %m\n");
        return 1;
    }
}

cap_t caps = cap_get_proc();        // Get the capability state of the process
if (caps == NULL) {                 // This returns all the different capability sets.
    perror("cap_get_proc");
    if (caps)
        cap_free(caps);
    return EXIT_FAILURE;
}

// STEP (2) Now from the above caps clear our 2 capabilities from the INHERITABLE set
int clear_inh_set = cap_set_flag(caps, CAP_INHERITABLE, num_caps, drop_caps, CAP_CLEAR);
if (clear_inh_set) {
    perror("cap_set_flag");
    cap_free(caps);
    return EXIT_FAILURE;
}

// Now set the cleared caps-structure as the processes new capabiity set.
int set_cap_set = cap_set_proc(caps);
if (set_cap_set) {
    perror("cap_set_proc");
    cap_free(caps);
    return EXIT_FAILURE;
}
cap_free(caps);
```

Use **prctl()** to drop the capabilities from the AMBIENT set
Use **cap_get_proc()** to get the capability sets of the process
Use **cap_set_flag()** to clear the capabilities from the INHERITABLE set
Use **cap_set_proc()** to set the cleared set back to the process

Use the approach shown above to complete the **setup_child_capabilities()** method in **sr_container_helpers.c**.

~~**You can test if this works by simply running *"mknod <SOME_NAME> p"*. If the capabilities have been set properly then this should fail.**~~

To test if the capabilities were set properly you can do the following:

- Copy the binary **'capsh'** found inside the **[/sbin]** folder of the docker container into the **[/sbin]** folder of the ***'rootfs'*** you downloaded to run containers.

  cp /sbin/capsh $ROOTFS/sbin/

- Now if you run **'capsh --print'** **[inside our SNR_CONTAINER]** without this method implemented *(i.e: capabilities not being filtered)* the output for **[Bounding set]** will indicate many capabilities.

- But after properly implementing this method *(filtering capabilities)* if you run the same command inside your SNR_CONTAINER container you will see a smaller set of capabilities for **[Bounding set]**

5. Disabling system calls inside a container

In addition to disabling capabilities, we also want to **restrict** processes inside our SRContainer from using certain **system-calls** that can possibly lead to a vulnerable state. **Seccomp** is one kernel feature which can be used to achieve this. This feature allows to control which system-calls a process and all its children have access to. It also enables to set the action to take *(kill the process, raise a signal, just allow it, etc)* when a process tries to execute such a system call. The intent is to allow untrusted processes to use the resources provided by the kernel with restricted access without abusing them .

In this assignment we will use this ***seccomp*** kernel feature to limit the system-calls allowed to the processes within our SRContainer. Support for the seccomp feature is provided by the libseccomp library.
The idea behind instigating this system-call restriction is as follows:

1. Create a system calls **filtering context** with a default behavior for all system-calls
2. Set up **filters** on this context for certain system calls that must be handled differently
3. Set any attributes that applies to created seccomp context.
4. Load the newly configured context into the kernel.
5. Release any memory allocated for the seccomp context that was just configured. This does not affect the context that was loaded into the kernel.

See the detailed description below of each of these steps.
(Trust me you can just use this as a one-to-one template to finish this part of the assignment)

STEP-1:

```
/**
 *  Initialize a seccomp context
 *      seccomp_init() is the method used
 *      SCMP_ACT_ALLOW - flag indicates that by default we want to ALLOW all system calls
 **/
scmp_filter_ctx seccomp_ctx = seccomp_init(SCMP_ACT_ALLOW);
if (!seccomp_ctx) {
    fprintf(stderr, "seccomp initialization failed: %m\n");
    return EXIT_FAILURE;
}
```

STEP-2:

```
/**
 *  Now we must set up filters for certain system calls for which we want different behavior
 *      Lets say we want to block the following system_calls. As in we want to kill any thread that tries to do them
 *          move_pages  - moves memory pages of a process to different blocks
 *          ptrace      - observe, track and control execution of another process
 *
 *          seccomp_rule_add() is the method used to add a filtering rule
 *          SCMP_FAIL is the action to take when there is match on this filter (IE. KILL the thread - no mercy!!!)
 *          SCMP_SYS(move_pages) -  the 3rd argument is the system-call number to which this filter applies.
 *                                  we use the SCMP_SYS() macro to get the correct number based on the underlying architecture
 *          4th argument - You use this argument if you dont want to capture all calls to the system call but want to capture
 *                         calls only when the system call has certain matching arguments.
 *                         Ex: the read() system call is called with the first argument being 0 (STDOUT)
 **/
int filter_set_status = seccomp_rule_add(
                                        seccomp_ctx,            // the context to which the rule applies
                                        SCMP_FAIL,          // action to take on rule match
                                        SCMP_SYS(move_pages),   // get the sys_call number using SCMP_SYS() macro
                                        0                       // any additional argument matches
                                    );
if (filter_set_status) {
    if (seccomp_ctx)
        seccomp_release(seccomp_ctx);
    fprintf(stderr, "seccomp could not add KILL rule for 'move_pages': %m\n");
    return EXIT_FAILURE;
}

filter_set_status = seccomp_rule_add(seccomp_ctx, SCMP_FAIL, SCMP_SYS(ptrace), 0);
if (filter_set_status) {
    if (seccomp_ctx)
        seccomp_release(seccomp_ctx);
    fprintf(stderr, "seccomp could not add KILL rule for 'ptrace': %m\n");
    return EXIT_FAILURE;
}
```

```
/**
 *  As another example lets say you want to disallow the 'unshare' system call
 *  But say that you want to disallow it ONLY if the 'CLONE_NEWUSER' flag is set in its argument
 *  Now we observe the signature for unshare() is as follows:
 *                  int unshare(int flags);
 *  Here flags can be an OR'ed combination of CLONE_FILES, CLONE_FS, CLONE_NEWCGROUP, CLONE_NEWUSER and many
 *  So we want to just capture calls to unshare() only if the 'CLONE_NEWUSER' flag is OR'ed in the argument.
 *  So seccomp_rule_add() would be as follows
 **/
filter_set_status = seccomp_rule_add(
                                    seccomp_ctx,                    // the context to which the rule applies
                                    SCMP_FAIL,                  // action to take on rule match
                                    SCMP_SYS(unshare),              // get the sys_call number using SCMP_SYS() macro
                                    1,                              // any additional argument matches
                                    SCMP_A0(SCMP_CMP_MASKED_EQ, CLONE_NEWUSER, CLONE_NEWUSER)
                                    );
if (filter_set_status) {
    if (seccomp_ctx)
        seccomp_release(seccomp_ctx);
    fprintf(stderr, "seccomp could not add KILL rule for 'unshare': %m\n");
    return EXIT_FAILURE;
}
```

In the last example (2nd image) we want to capture calls to unshare() only if the 'CLONE_NEWUSER' flag is used.

So in the call to **seccomp_rule_add(),** in its 4th argument we say we want "one" argument match on the call to unshare.

We include what this match is in the 5th argument of **seccomp_rule_add().**

```
SCMP_A0(SCMP_CMP_MASKED_EQ, CLONE_NEWUSER, CLONE_NEWUSER)
```

`SCMP_A0` -

Tells to match the 0th argument of unshare(). If it was SCMP_A1 then the match must be on 1st argument. Notice that its 0 indexed like arrays in C

`SCMP_CMP_MASKED_EQ` -

Tells that it's not a one to one match but its a check on a MASKED argument. This is because the argument to unshare() can be an OR of many flags: CLONE_FS | CLONE_FILES | CLONE_VM | CLONE_NEWUSER.

3rd Argument: The mask for validation
4th Argument: What it must be equal to

So similarly you can write rules to match certain arguments on the system_call when filtering them.

STEP-3:

```c
/**
 *  Set the 'SCMP_FLTATR_CTL_NNP' attribute on the newly created context
 *  This attribute is used to ensure that the NO_NEW_PRIVS functionality is enabled
 *  This is to control previledge escalation of child processes spawned with exec() in a parent with lesser priviledge
 *  You can just copy this and use it at the end of all seccomp rules.
 **/
filter_set_status = seccomp_attr_set(seccomp_ctx, SCMP_FLTATR_CTL_NNP, 0);
if (filter_set_status) {
    if (seccomp_ctx)
        seccomp_release(seccomp_ctx);
    fprintf(stderr, "seccomp could not set attribute 'SCMP_FLTATR_CTL_NNP': %m\n");
    return EXIT_FAILURE;
}
```

Set the filter attribute value of `SCMP_FLTATR_CTL_NNP`.

STEP-4:

```c
/**
 *  Finally load the created context into the kernel and release it from the current process memory
 **/
filter_set_status = seccomp_load(seccomp_ctx);
if (filter_set_status) {
    if (seccomp_ctx)
        seccomp_release(seccomp_ctx);              // release from current process memory.
    fprintf(stderr, "seccomp could not load the new context: %m\n");
    return EXIT_FAILURE;
}
```

Load the created context into the kernel. You can simply re-use Step-3 and 4 as is.

Your task ***(should you choose to accept it)*** is to:

Complete the method **`setup_syscall_filters()`** in the **sr_container_helpers.c** file to **STOP** our SRContainer from invoking the following system calls. Any process that attempts to run these system calls must be killed (~~**`SCMP_ACT_KILL`**~~ **`SCMP_FAIL`**). All other system calls must be allowed.

- ptrace
- mbind
- migrate_pages
- move_pages
- unshare (***Only restrict if the CLONE_NEWUSER flag is used***)
- clone (***Only restrict if the CLONE_NEWUSER flag is used)***
- chmod (***Only restrict if the `S_ISUID or S_ISGID` flags are used for the "mode" argument)***

You can test if this works by simply writing a C program which tries to use one of the system calls above.

Instructions to copy your code into the host-container environment.

You must first copy the template code folder "A3Template" to the cs310 server.

scp -r <path_in_your_pc>/A3Template <socs_uname>@cs310.cs.mcgill.ca:~

You can compile the program by simply running **'make container'** with the given *Makefile* or use the complete **'gcc'** command:

gcc -o SNR_CONTAINER -g -Wall -Werror sr_container.c sr_container_helpers.c sr_container_utils.c -lseccomp -lcap

Then, you must copy the built executable into your own docker-container environment. That is, the container you created with **'docker run'** for Phase-1.

docker cp ~/A3Template/SNR_CONTAINER <container_name>:/home

Now, if you go into your container using:

docker exec -it <container_name> /bin/bash

You should see your executable and you can run it with the correct flags.

Do not copy your entire *A3Template* code into your docker container. Only copy the built executable.

What to submit:

sr_container.c (with your changes)

sr_container_helpers.c (with your changes)

No need to submit any other files since you will not have to change them.

Rubric (This Phase accounts for 40%)

1. Setting up cgroups/hostname flags: 7%
2. Implementing child process logic: 7%
3. Proper usage of **pivot_root()**: 6%
4. Implementing capabilities: 10%
5. Implementing syscall filtering: 10%