Labs for Container Security From the Bottom Up

In your VM, the username is ‘class’ and the password is ‘classy’

Mount Namespace Lab 1

The Mount Namespace labs will take you through a set of commands you can execute in a terminal to gain an understanding of the ‘Mount’ namespace. Mount is a fundamental namespace and required in order to achieve a usable level of isolation in container ecosystems

This first lab demonstrates that basic usage of ‘unshare -mr’ doesn’t provide any isolation from the existing host mount namespaces.

1. **Open the Terminal in your Lab VM**
    a. Make two tabs or windows, these will be referred to as “left” and “right”

2. **Right Terminal: lsns**
    a. lsns (list namespaces) prints out information about existing namespaces
    b. You should have one of each type. Cgroup, pid, user, uts, ipc, mnt, and net.
    c. Why so few? You are not root, so you can’t see them all.

3. **Left Terminal: unshare -mr**
    a. You are now inside a new mount namespace
    b. Note that you suddenly appear to be root (at least, bash tells you that) despite having run this command without a sudo

4. **Right Terminal: lsns**
    a. You’ll see not one but *two* new namespaces were created, a mount namespace (from -m) and also a user namespace.
    b. The user namespace governs permissions to all other namespaces, thus a new user namespace is created to control this new mount namespace.
    c. The above -r option in the unshare command configures this new user namespace such that your user is root
    d. Optionally, you could run `unshare -Um` instead of step 3. Your user will be ‘nobody’ instead of root.

5. **Left Terminal: ls /**
    a. When you create a new mount namespace from the base context, mounts are automatically shared with you. Thus, you can still see all system files.

6. **Left Terminal: touch /tmp/namespaceWasHere**
    a. No error - you can write to /tmp just like the class user

7. **Both Terminals: ls /tmp**
    a. The /tmp/namespaceWasHere file was created, and is visible both in your user's new namespace and the original pre-unshare namespace

8. **Both Terminals: ls /root**
    a. Same error in both - permission denied
    b. You may appear to be root in the new namespace which unshare created for you, but the overall system knows that you are not root and won't let you perform root-level actions

9. **Left Terminal: mount**
    a. Quite a lot of mounts are propagated to the new mount namespace by default

10. **Left Terminal: mount -t tmpfs tmpfs /mnt**
    a. This creates a tmpfs mount onto the /mnt directory in your mount namespace

11. **Both Terminals: mount | grep mnt**
    a. In the left window you will see an entry such as:
    ```
    tmpfs on /mnt type tmpfs (rw,relatime,uid=1000,gid=1000,inode64)
    ```
        i. This is the temp filesystem which was just mounted
    b. In the right window you will either see nothing or an entry about nsfs - but not the new tmpfs mount at /mnt

This lab demonstrated that a lot of mount points are still shared with you after usage of unshare -mr, while also showing that mounts created in the new mount namespace are not visible to an unprivileged host user (root can see them all, of course).

Mount Namespace Lab 2

This lab demonstrates how to achieve a level of root '/' filesystem hiding while also giving your target process some isolation from the system at large - all thanks to namespaces.

1. **Big surprise, open the terminal (Two, left and right)**
    a. If reusing a a 'left terminal' from the previous exercise, run 'exit' to return to the class user context

2. **Left Terminal: wget http://dl-cdn.alpinelinux.org/alpine/v3.15/releases/x86_64/alpine-minirootfs-3.15.0-x86_64.tar.gz**
    a. This is Alpine Linux's distribution of a minimal root filesystem, containing basic default binaries that can be used as a base to build on.
    b. ** If for some reason network connectivity is poor, this archive has been saved to **~/Desktop/labs/mount2**
        i. In this case you would:
        ii. **cp ~/Desktop/labs/mount2/alpine-minirootfs-3.15.0-x86_64.tar.gz ~**
        iii. And then continue on with the instructions as written

3. **Left Terminal: mkdir alpineroot && tar -xzf alpine-minirootfs-3.15.0-x86_64.tar.gz -C alpineroot**
    a. To demonstrate how minimal this alpine root filesystem is, compare the output of \`ls alpineroot/bin\` (alpine linux bin folder) to \`ls /bin\` (ubuntu /bin folder)
    b. Same for:
        i. alpine's /lib vs ubuntu /lib
    c. Alpine Linux really is a teeny tiny distribution

4. **Left Terminal: cd alpineroot**

5. **Left Terminal: chown class . -R**
    a. This ensures that the class user is marked as owner of all the files which were extracted from the alpine linux tarball, just to be sure

6. **Left Terminal: touch etc/resolv.conf && echo 'nameserver 8.8.8.8' > etc/resolv.conf**
    a. This step is necessary because alpine linux doesn't come with a default nameserver configuration, and this was causing me all sorts of errors while working on these instructions

7. **Left Terminal: unshare -Ufmrp --mount-proc --root=/home/class/alpineroot --wd=/ sh**
    a. This is doing a whole bunch of things all at the same time
    b. **-U** creates a new user namespace
        i. Note that **-U** is implied by **-r**
        ii. **-r** auto-maps your 'class' user into 'root' in the new namespace for convenience
    c. **-m** creates the new mount namespace
    d. **-p** creates a new PID namespace, and **--mount-proc** mounts a /proc which is inside that new PID namespace. Note that this also implies **-m**.
    e. **-f** is an important flag. It first forks off a new process, and then runs your target executable (in this case, **sh**) inside that process. The forked process becomes PID 1, which serves as the init process and *must* remain running or your namespace will crash, effectively.

        i. Without **-f**, ‘**sh**’ will run some subprocesses on startup and the first one of those will be PID 1. After that subprocess returns and exits, PID 1 no longer exists and you get interesting errors like ‘cannot allocate memory’

   f. --root
      i. This was probably the most interesting and useful capability of unshare that I discovered while researching this presentation
      ii. **--root** is effectively a pivot_root operation, specifying the ‘alpineroot’ directory as the new ‘/’ point
      iii. Using this argument allows you to avoid setting up a bind mount and going through the pivot_root, unmount old operation

   g. --wd
      i. Provides a nice shortcut to say “put execution at / after stepping into the new root”

   h. And finally, **sh**
      i. Alpine Linux minimal rootfs doesn’t come with bash, so ‘**sh**’ it is

8. Left Terminal: ps -ef

   a. Your results will look similar to:

```
PID   USER     TIME  COMMAND
   1 root      0:00 sh
   4 root      0:00 ps -ef
```

   b. Thanks to the PID namespace isolation, your `sh` process can’t see other processes owned by the ‘class’ user or anything else in the system

   ** PID has it’s own lab coming up… but you see the gist of it here!

9. Left Terminal: mount

   a. In the namespace-isolated terminal, you get:

```
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

   b. No other mounts are visible
   c. Feel free to run this command in the non-isolated terminal as a reminder of how many mount points there are by default (so many)

10. Right Terminal: lsns

   a. This is just a quick command for you to see the namespaces which were created by the unshare command in this case. User, Mount, and PID. In my case:

```
        4026532758 user        2  5191 user unshare -Ufmrp --mount-proc
--root=/home/class/Desktop/labs/mount2/alpineroot
        4026532759 mnt         2  5191 user unshare -Ufmrp --mount-proc
--root=/home/class/Desktop/labs/mount2/alpineroot
        4026532760 pid         1  5192 user sh
```

   b. Note that the user and mnt namespaces list the unshare command, but PID lists not unshare but ‘sh’. This is because sh is the init process (PID 1) in that namespace.

This lab demonstrates one way in which you can isolate a process: Set up a root directory, create new namespaces for it, pivot the root into that directory, and let the process run amok.

PID Namespace Lab 1

PID namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs.

In this lab you will set up a new PID namespace along with an associated **/proc** mount-point to enable use of the namespace by tools running within it. Given the prevalence of /proc-usage among core linux utilities, it's basically required to combine both a mount as well as a PID namespace to get value from this capability.

Not mentioned here (yet still important) is IPC namespace: IPC allows processes to communicate with each other and for improved isolation you'd want to combine an IPC, PID and MOUNT namespace together.

Like the previous labs on the Mount namespace, we will be leveraging the **unshare** and **nsenter** commands as it is readily available on modern linux distributions and allows us to slice-and-dice our way into new namespaces. At the end of this lab you should have a good sense of what the PID namespace does in practice

1. **Again, you need two Terminals (Two, left and right)**
    a. If reusing a a 'left terminal' from the previous exercise, run 'exit' to return to the class user context

2. **Left Terminal: unshare -mrfp --mount-proc**
    a. This feels pretty similar to a previous exercise, and it is. We're making a new pid namespace and auto-mounting /proc over the top of the one which already exists in a new mount namespace. And in the new namespace your user is mapped as root.

3. **Left Terminal: sleep 1h 1m &**
    a. The ampersand in this command is important, or your terminal will take a rather long nap
    b. The point of this is to set up a process in this PID namespace that we can inspect

4. **Left Terminal: ps -ef**

   a. You will see your new sleep process and its arguments, but not the outer system pids and processes. Simple enough, right?

5. **Left Terminal: unshare -mrfp --mount-proc** (time for PID Inception)
   a. Poof - you have a new PID namespace which is **nested** inside the other PID namespace. It's not a standalone namespace, as PID namespaces are all nested from the root context.

6. **Left Terminal: ps -ef**
   a. As you might expect, you can't see the sleep process which we just ran a minute ago. This PID namespace is just as isolated as the previous one.

7. **Left Terminal: sleep 1h 10m &**
   a. Setting up for more process inspection in a minute

8. **Left Terminal: unshare -mrfp --mount-proc**
   a. Yep, more nesting

9. **Left Terminal: ps -ef**
   a. OK, that's enough nesting for now. This terminal is now nested a few levels deep in PID namespaces. But what use is this?

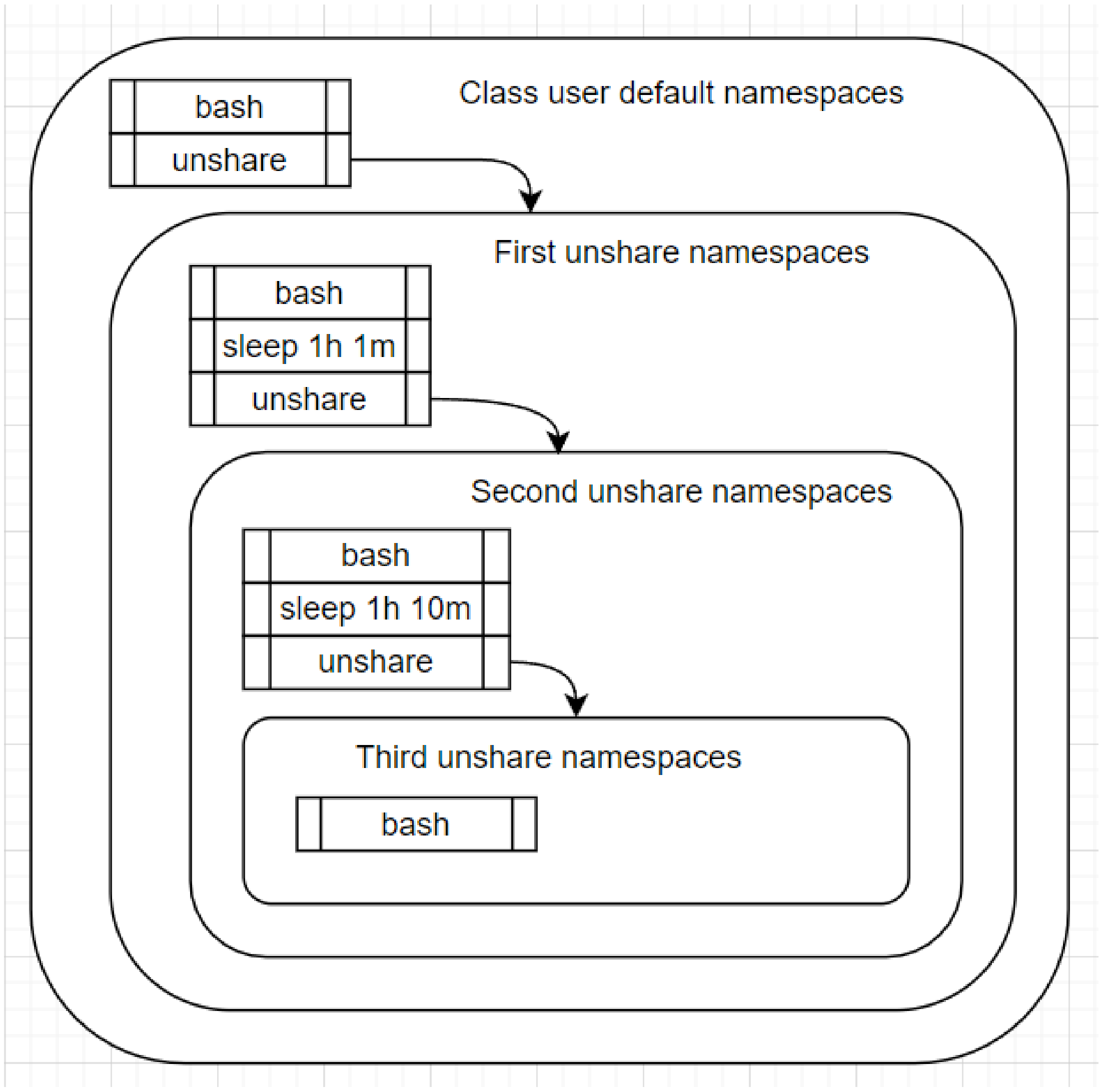
10. **Left Terminal: sleep 1h 15m &**
    a. Yes, another sleep process for inspection. We're about to get to that part.

11. **Right Terminal: lsns -t pid**
    a. lsns is the tool for listing namespaces. Your right-hand terminal window is in the 'class' user context. This is the parent context for all of the PID namespaces you just created, and can see all of them.

    b. Example output:
```
        NS       TYPE NPROCS   PID   USER    COMMAND
4026531836 pid      65         1542    class   /lib/systemd/systemd --user
4026532634 pid       3         2265    class   -bash   note: NS ID 1
4026532637 pid       3         2277    class   -bash   note: NS ID 2
4026532640 pid       2         2289    class   -bash   note: NS ID 3
```

    c. We've created three PID namespaces, and all are shown here as running the -bash command (which is what unshare defaults to)
    d. The topmost listing showing systemd is your 'class' user level context
    e. Now let's inspect one of these namespaces. Pick the topmost namespace which is shown as command -bash and note its ID (in this case, NS 4026532634)

12. **This is a graphic showing what we've set up**



13. Right Terminal: ps -e -o pidns,pid,args | grep <NS ID Number 1>

a. In the output you will see something like this:
```
4026532634    2265 -bash
4026532634    2274 sleep 1h 1m
4026532634    2276 unshare -mrpf --mount-proc
```
b. Why can you see these processes from the 'class' user context? Simple. Your 'class' user owns the PID namespace and can see all processes inside it.

c. Let's look at the next-level nested bash - think you'll be able to see those processes?

d. Take the NS ID of the second level unshare and do the same thing:

14. Right Terminal: ps -e -o pidns,pid,args | grep <NS ID Number 2>

a. Use the second level unshare PID namespace ID here (see the notes which are red above the graphic)
b. Example output:
```
4026532637    2277 -bash
4026532637    2286 sleep 1h 10m
4026532637    2288 unshare -mrpf --mount-proc
```
c. The ‘class’ user can see into this namespace too, because it’s nested under one which is owned by ‘class’.
d. We’re going to look at a tool called ‘nsenter’ which is used to step a process into a set of namespaces. But first we need to know how to set it up.

15. Right Terminal: ps -ef | grep unshare

a. Example output (header added by me for clarity)
```
UID        PID    PPID  C STIME TTY        TIME CMD
class      2033    1662  0 06:15 pts/0    00:00:00 unshare -mrfp --mount-proc (#1)
class      2044    2034  0 06:15 pts/0    00:00:00 unshare -mrfp --mount-proc (#2)
class      2055    2045  0 06:15 pts/0    00:00:00 unshare -mrfp --mount-proc (#3)
```

b. These are the **PIDs** and **Parent PIDs** for the unshare processes you have executed. As a reminder, the ‘unshare’ process does not change its namespace when executed. Unshare sets up the target process to run in a different namespace.
c. PID 2033 here corresponds with the first unshare which was run in step 2, so it’s currently in the base ‘class’ user namespace set.
d. PID 2044 (PPID 2034) here corresponds to the second unshare (step 5)
e. PID 2055 (PPID 2045) here corresponds to the third unshare (step 8)

16. Right Terminal: sudo nsenter --target <pid namespace PPID> -p -m ps -ef

a. **For pid namespace PPID, use #2 labeled above in 15a (2034 in my case)**

As an example, here is a complete **nsenter** command for the above example:
sudo nsenter --target 2034 -p -m ps -ef
** Remember that ‘2034’ is the **Parent PID** of our first namespace
And this is the **_second_** line in the ps-ef output
Depending on output formatting, you may need the 2nd or 3rd entry

b. **nsenter** is a tool for stepping into a target namespace and running a process. In this case, we’re stepping into the pid and mount namespaces of the unshare command itself (**-p -m**) and then running **ps -ef**

c. Example output:
```
UID          PID    PPID  C STIME TTY          TIME CMD
class          1       0  0 04:27 pts/0    00:00:00 -bash
class          9       1  0 04:27 pts/0    00:00:00 sleep 1h 1m
class         13       1  0 04:27 pts/0    00:00:00 unshare -mrfp --mount-proc
class         14      13  0 04:27 pts/0    00:00:00 -bash
class         24      14  0 04:27 pts/0    00:00:00 sleep 1h 10m
class         26      14  0 04:28 pts/0    00:00:00 unshare -mrfp --mount-proc
class         27      26  0 04:28 pts/0    00:00:00 -bash
class         37      27  0 04:28 pts/0    00:00:00 sleep 1h 15m
root          42       0  0 04:33 pts/1    00:00:00 ps -ef
```

d. This ps -ef command shows all three sleeps and the two ‘inner’ unshares. But why does it show them all? Because we stepped into the namespaces which the unshare process itself is running in. Unshare is in the same PID namespace as the bash which ran the first sleep, and thus can see it.

e. Note that when you were running the sleep and unshare commands, your left terminal window showed ‘root’ - but in this view we see it was ‘class’. Linux knew who you were the whole time, even if your view was a little different.

17. Right Terminal: sudo nsenter --target <pid namespace PPID, step 14 #3> -p -m ps -ef

a. **For pid namespace PPID, use #3 labeled above in 14a (2045 in this case)**
   You may need to use the second entry, depending on output formatting
   i. **Example full command: sudo nsenter --target 2045 -p -m ps -ef**

b. We’re now running PS in the middle unshare

c. Example output:
```
UID          PID    PPID  C STIME TTY          TIME CMD
class          1       0  0 04:27 pts/0    00:00:00 -bash
class         11       1  0 04:27 pts/0    00:00:00 sleep 1h 10m
class         13       1  0 04:28 pts/0    00:00:00 unshare -mrfp --mount-proc
class         14      13  0 04:28 pts/0    00:00:00 -bash
class         24      14  0 04:28 pts/0    00:00:00 sleep 1h 15m
root          26       0  0 04:35 pts/1    00:00:00 ps -ef
```

d. As expected, the middle unshare PID namespace can see its own processes and those of its child PID namespace

Extra Credit: As an additional exercise, from the right terminal find the PIDs for your bash processes and step into those namespaces. This lab used nsenter on the unshare process, which has a different view than the bash processes.

Network Namespace Lab 1

This pair of labs will demonstrate the network isolation features associated with Networking. If you recall, we need both the 'Net' and 'UTS' namespaces in order to set up an environment where your process(es) can have their own IP address and hostname. When you get right down to it, the UTS namespace has little utility for us. We mention it here for completeness and to let the class ponder over why such a namespace exists at all(?)

Lab 1 uses slirp4netns which is pre-installed on the lab vm. Lab 2 uses more traditional linux network commands and is more complex

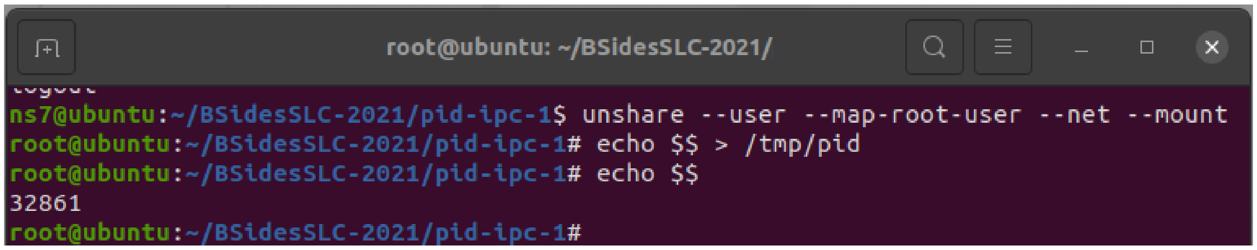
By the end of lab 1 you will have carved out your own isolated network environment that has its own IP address and hostname. You will be able to 'reach out' to the internet from within the environment. **slirp4netns** is used for simplicity in providing a network connection 'out' to the internet

1. You will need to open **Two terminal windows** to successfully accomplish this lab

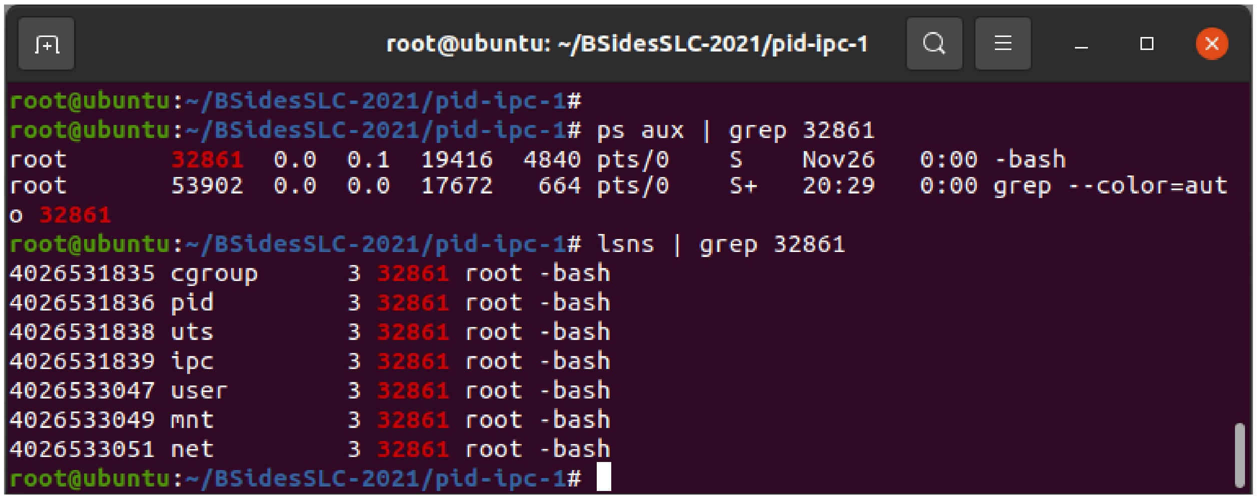
2. In the **first window**:
   a. unshare --user --map-root-user --net --uts --mount
      i. This configures a few new namespaces and deposits you into them:
         1. User
         2. Network
         3. UTS (Unix Time Sharing … Allows for unique host/domain names)
         4. Mount

   b. echo $$
      i. This shows the bash PID that is assigned to the new namespaces created by the 'unshare' command
   c. lsns | grep <PID FROM 'b' ABOVE>
      i. lsns will show the namespaces created and the process(es) they are associated with

   You should see something like this:



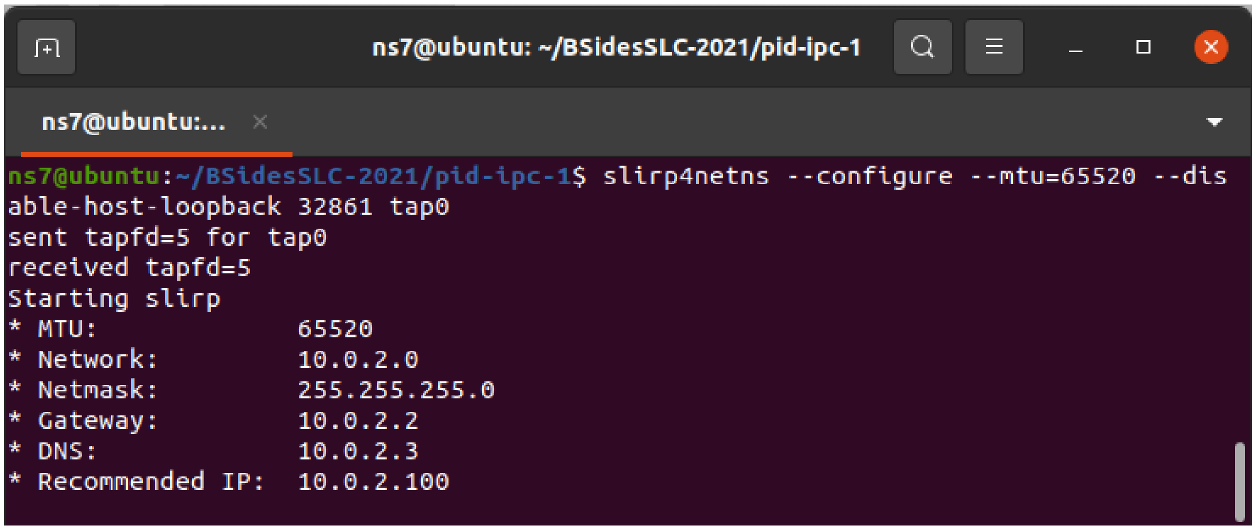

```
ns7@ubuntu:~/BSidesSLC-2021/pid-ipc-1$ unshare --user --map-root-user --net --mount
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# echo $$ > /tmp/pid
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# echo $$
32861
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1#
```



```
root@ubuntu: ~/BSidesSLC-2021/pid-ipc-1
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1#
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# ps aux | grep 32861
root       32861  0.0  0.1  19416  4840 pts/0    S    Nov26   0:00 -bash
root       53902  0.0  0.0  17672   664 pts/0    S+   20:29   0:00 grep --color=aut
o 32861
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# lsns | grep 32861
4026531835 cgroup      3 32861 root -bash
4026531836 pid         3 32861 root -bash
4026531838 uts         3 32861 root -bash
4026531839 ipc         3 32861 root -bash
4026533047 user        3 32861 root -bash
4026533049 mnt         3 32861 root -bash
4026533051 net         3 32861 root -bash
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1#
```

These commands show that namespaces were created and associated with the bash process spawned by the unshare command

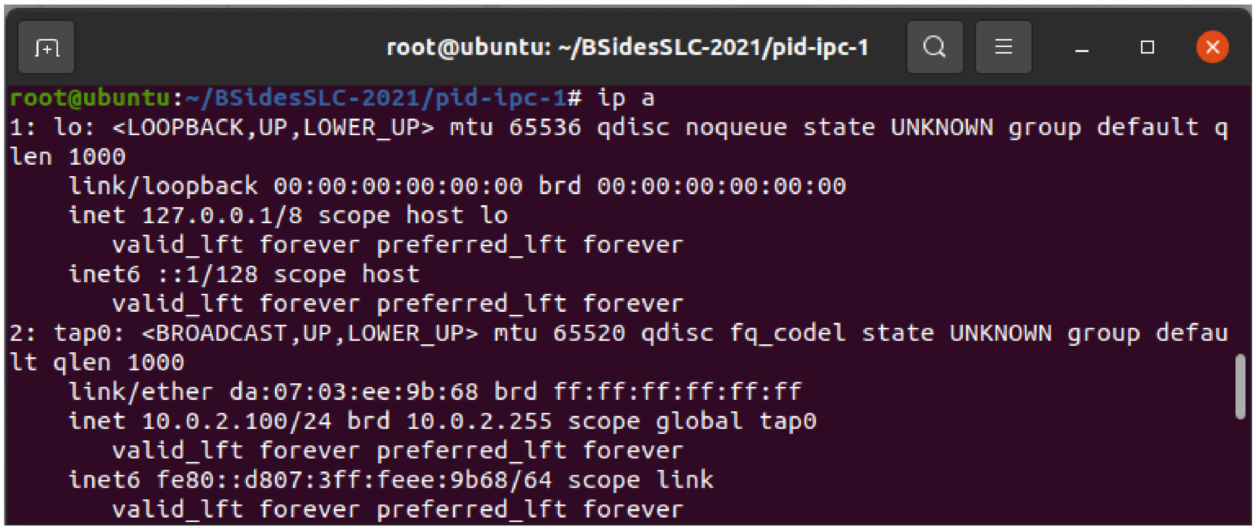
3. In the **second terminal window**, execute this command:
    a. slirp4netns --configure --mtu=65520 --disable-host-loopback <PID FROM Step 3.b above> tap0



```
ns7@ubuntu: ~/BSidesSLC-2021/pid-ipc-1
ns7@ubuntu:...
ns7@ubuntu:~/BSidesSLC-2021/pid-ipc-1$ slirp4netns --configure --mtu=65520 --dis
able-host-loopback 32861 tap0
sent tapfd=5 for tap0
received tapfd=5
Starting slirp
* MTU:             65520
* Network:         10.0.2.0
* Netmask:         255.255.255.0
* Gateway:         10.0.2.2
* DNS:             10.0.2.3
* Recommended IP:  10.0.2.100
```

slirp4netns lets us configure networking without root permissions and is an easy way for us to practice with network namespaces.

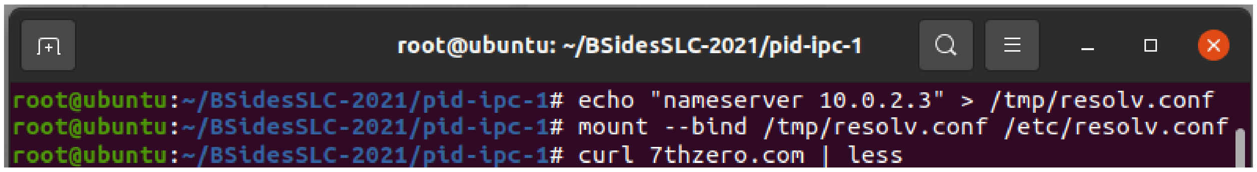
4. Return to your **first terminal window** and verify that networking is enabled:
    a. ip a



```
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default q
len 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tap0: <BROADCAST,UP,LOWER_UP> mtu 65520 qdisc fq_codel state UNKNOWN group defau
lt qlen 1000
    link/ether da:07:03:ee:9b:68 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.100/24 brd 10.0.2.255 scope global tap0
       valid_lft forever preferred_lft forever
    inet6 fe80::d807:3ff:feee:9b68/64 scope link
       valid_lft forever preferred_lft forever
```

This shows us that there is a **tap0** interface configured

5. Using the information available in **step 3.a**, configure the networking within the network namespace:
    a. echo "nameserver 10.0.2.3" > /tmp/resolv.conf
    mount --bind /tmp/resolv.conf /etc/resolv.conf
    curl 7thzero.com | less

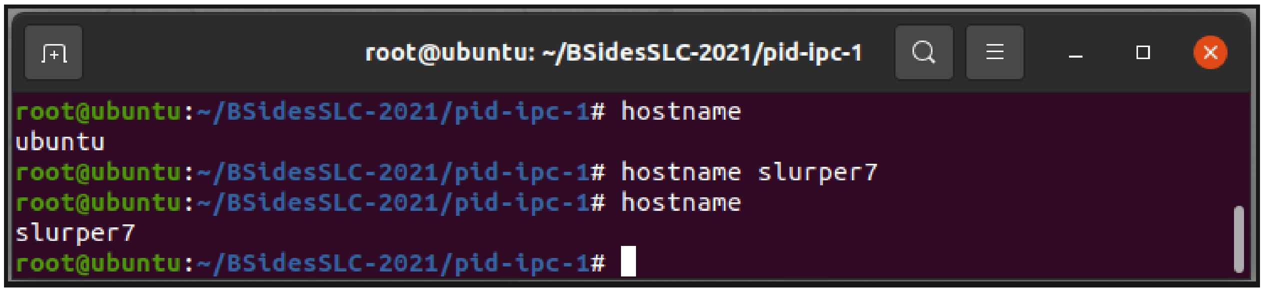


```
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# echo "nameserver 10.0.2.3" > /tmp/resolv.conf
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# mount --bind /tmp/resolv.conf /etc/resolv.conf
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# curl 7thzero.com | less
```

These commands configure dns resolution within the namespace and allow you to make network calls ‘outside your namespace’ to the outside world, courtesy of slirp4netns.

Feel free to ping/curl your way around the internet!

6. Perfunctory use of the UTS namespace to change the hostname:
    a. hostname slurper7
       hostname



```
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# hostname
ubuntu
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# hostname slurper7
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1# hostname
slurper7
root@ubuntu:~/BSidesSLC-2021/pid-ipc-1#
```

As you can see, the hostname associated with this process is changed from the default to the manually-assigned entry.

Network Namespace Lab 2

Lab 2 takes things a step further and has you setup 2 network namespaces, each operating as their own distinct network identity. You will set up network rules that allow the network namespaces to interact with each other using their designated IP address ranges.

For this to work, **please open 3 separate terminals**

1. **Within terminal 1 (which is our 'host' terminal):**
   **** This sets up the network namespaces and virtual networks/interfaces**

   Configure virtual network adapters:
    a. ip link add veth0 type veth peer name veth1
    b. ip link add veth10 type veth peer name veth11

2. Create network namespaces for **veth0** and **veth10** (named **vnet0** and **vnet10**)
    a. ip netns add vnet0
    b. ip netns add vnet10

3. Assign the **veth0** and **veth10** interfaces to their respective namespaces
    a. ip link set veth0 netns vnet0
    b. ip link set veth10 netns vnet10

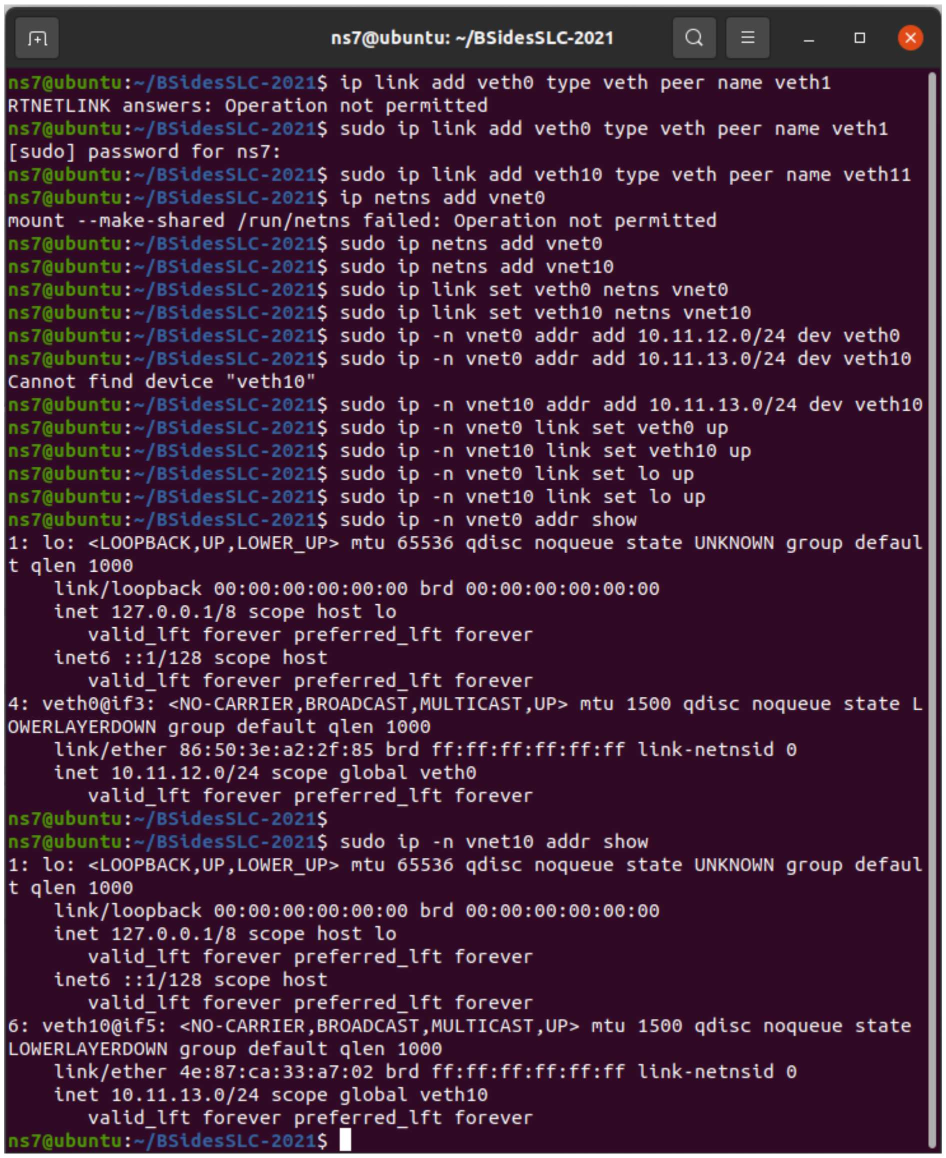
4. Configure IP ranges for each interface
    a. ip -n vnet0 addr add 10.11.12.0/24 dev veth0
    b. ip -n vnet10 addr add 10.11.13.0/24 dev veth10

5. Bring up each interface
    a. ip -n vnet0 link set veth0 up
    b. ip -n vnet10 link set veth10 up

6. Bring up '**lo**' interfaces for each virtual network to facilitate the 'local' route table
    a. ip -n vnet0 link set lo up
    b. ip -n vnet10 link set lo up

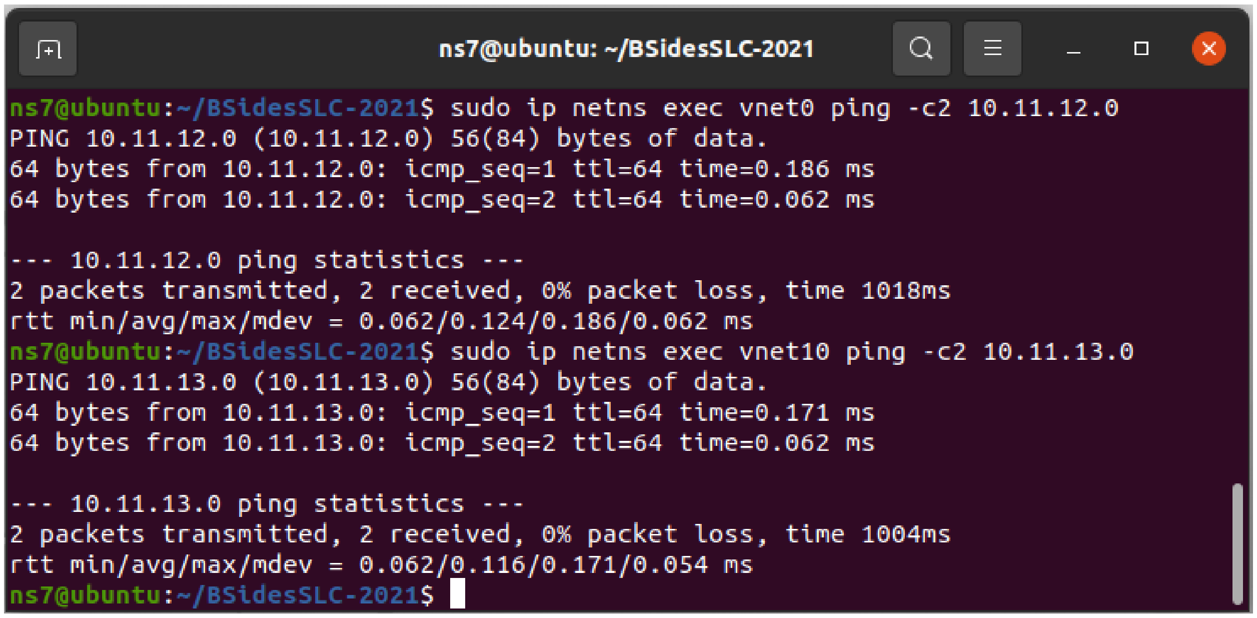
7. Confirm that the interfaces are 'up' and available:
    a. ip -n vnet0 addr show
    b. ip -n vnet10 addr show

Here's what you could see from the above operations:



```
ns7@ubuntu: ~/BSidesSLC-2021
ns7@ubuntu:~/BSidesSLC-2021$ ip link add veth0 type veth peer name veth1
RTNETLINK answers: Operation not permitted
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip link add veth0 type veth peer name veth1
[sudo] password for ns7:
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip link add veth10 type veth peer name veth11
ns7@ubuntu:~/BSidesSLC-2021$ ip netns add vnet0
mount --make-shared /run/netns failed: Operation not permitted
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns add vnet0
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns add vnet10
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip link set veth0 netns vnet0
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip link set veth10 netns vnet10
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 addr add 10.11.12.0/24 dev veth0
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 addr add 10.11.13.0/24 dev veth10
Cannot find device "veth10"
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 addr add 10.11.13.0/24 dev veth10
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 link set veth0 up
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 link set veth10 up
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 link set lo up
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 link set lo up
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: veth0@if3: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state L
OWERLAYERDOWN group default qlen 1000
    link/ether 86:50:3e:a2:2f:85 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.11.12.0/24 scope global veth0
       valid_lft forever preferred_lft forever
ns7@ubuntu:~/BSidesSLC-2021$
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
6: veth10@if5: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
LOWERLAYERDOWN group default qlen 1000
    link/ether 4e:87:ca:33:a7:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.11.13.0/24 scope global veth10
       valid_lft forever preferred_lft forever
ns7@ubuntu:~/BSidesSLC-2021$
```

8. Check/verify that we can run arbitrary commands in the network namespaces:
    a. ip netns exec vnet0 ping -c2 10.11.12.0
    b. ip netns exec vnet10 ping -c2 10.11.13.0



```
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns exec vnet0 ping -c2 10.11.12.0
PING 10.11.12.0 (10.11.12.0) 56(84) bytes of data.
64 bytes from 10.11.12.0: icmp_seq=1 ttl=64 time=0.186 ms
64 bytes from 10.11.12.0: icmp_seq=2 ttl=64 time=0.062 ms

--- 10.11.12.0 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1018ms
rtt min/avg/max/mdev = 0.062/0.124/0.186/0.062 ms
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns exec vnet10 ping -c2 10.11.13.0
PING 10.11.13.0 (10.11.13.0) 56(84) bytes of data.
64 bytes from 10.11.13.0: icmp_seq=1 ttl=64 time=0.171 ms
64 bytes from 10.11.13.0: icmp_seq=2 ttl=64 time=0.062 ms

--- 10.11.13.0 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 0.062/0.116/0.171/0.054 ms
ns7@ubuntu:~/BSidesSLC-2021$
```

The respective pings should function within their own network. At present they should not work between network namespaces for lack of routing

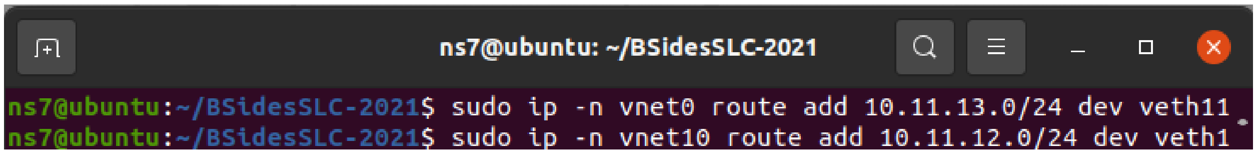
You can run other commands like **tcpdump** in the target network namespace using **ip netns exec <network-namespace> <command>**

9. Configure 'return' veth adapters into each network namespace
    a. sudo ip link set veth1 netns vnet10
    b. sudo ip link set veth11 netns vnet0
    c. sudo ip -n vnet0 link set veth11 up
    d. sudo ip -n vnet10 link set veth1 up

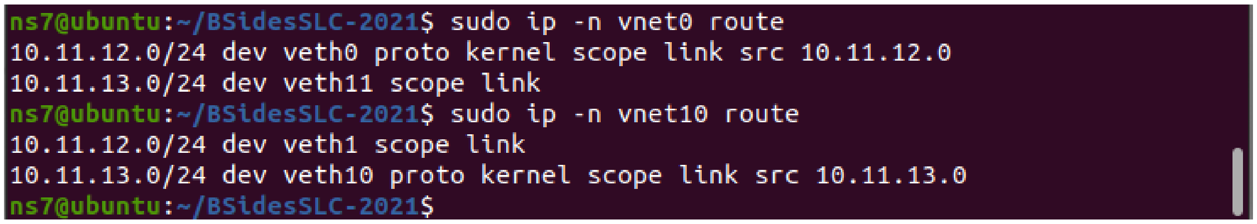
    ** This ensures that each veth pair is connected between network namespaces, which is useful for bi-directional communication.

    ...For the astute observer: yes, we could have simply used veth0 and veth1 to connect the namespaces as well.

10. Configure routing between network namespaces
    a. sudo ip -n vnet0 route add 10.11.13.0/24 dev veth11
    b. sudo ip -n vnet10 route add 10.11.12.0/24 dev veth1



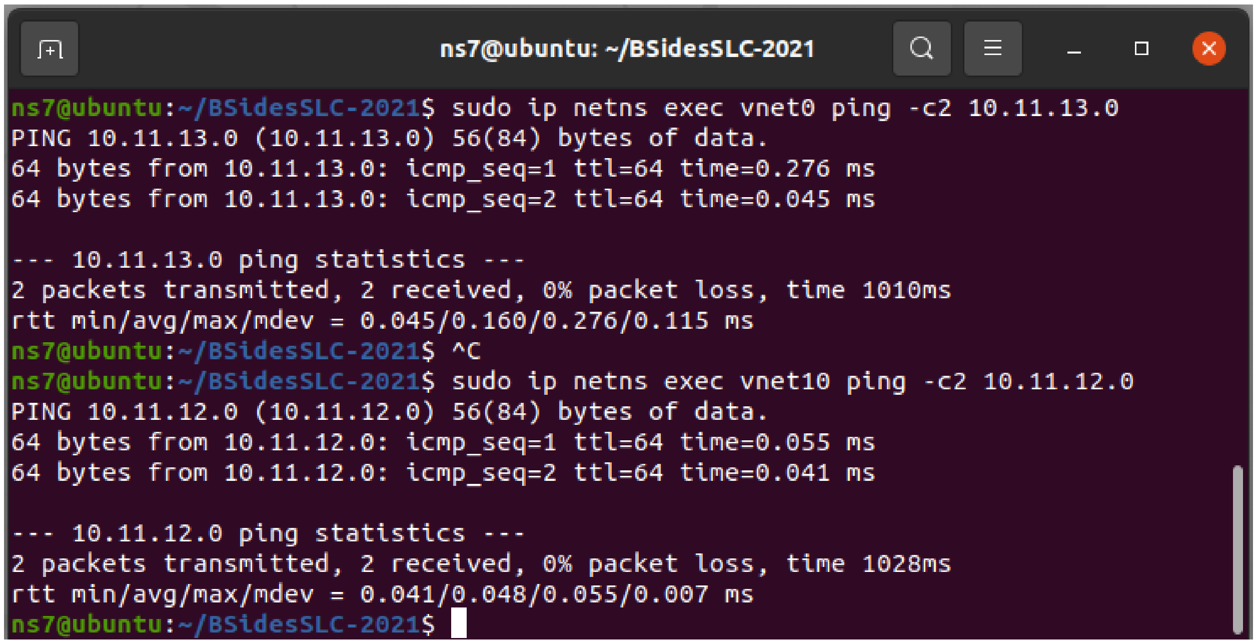
```
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 route add 10.11.13.0/24 dev veth11
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 route add 10.11.12.0/24 dev veth1
```



```
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet0 route
10.11.12.0/24 dev veth0 proto kernel scope link src 10.11.12.0
10.11.13.0/24 dev veth11 scope link
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip -n vnet10 route
10.11.12.0/24 dev veth1 scope link
10.11.13.0/24 dev veth10 proto kernel scope link src 10.11.13.0
ns7@ubuntu:~/BSidesSLC-2021$
```

A note on deleting routes: If you need to delete a route, use a command like this:
sudo ip -n vnet0 route del 10.11.12.0/24 dev veth10 scope link
** You can run sudo ip -n vnet0 route to view the list of routes that can be deleted

11. Attempt to ping each subnet from the other namespace leveraging ip netns exec
    a. sudo ip netns exec vnet0 ping -c2 10.11.13.0
    b. sudo ip netns exec vnet10 ping -c2 10.11.12.0

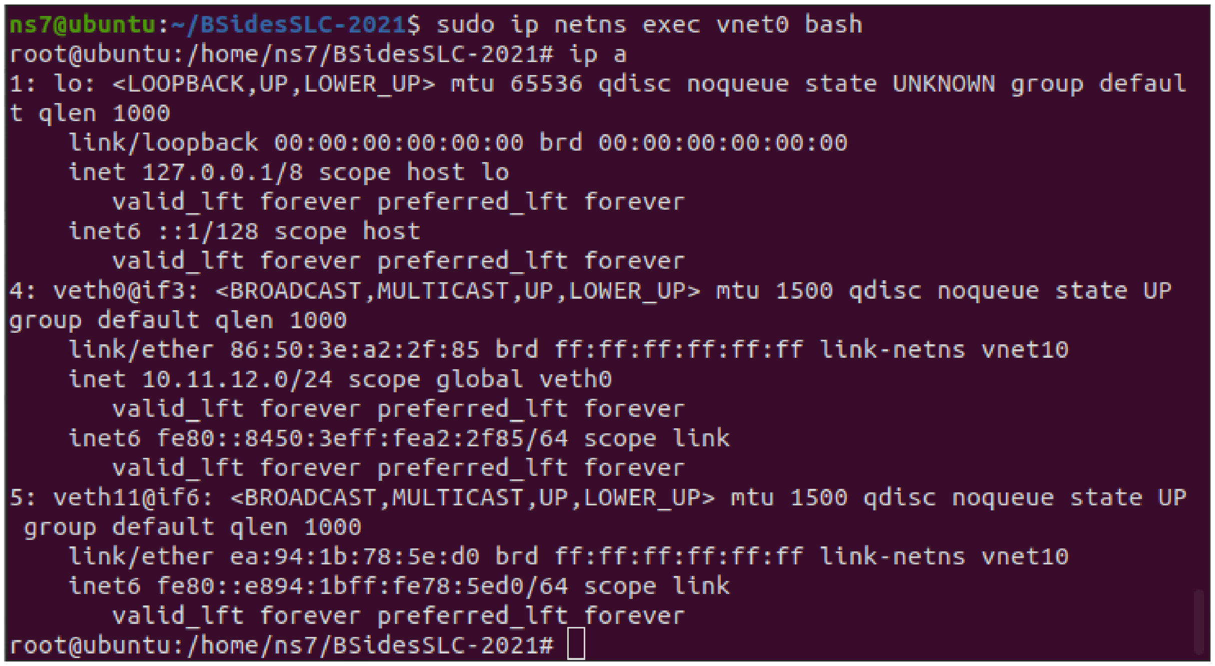


```
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns exec vnet0 ping -c2 10.11.13.0
PING 10.11.13.0 (10.11.13.0) 56(84) bytes of data.
64 bytes from 10.11.13.0: icmp_seq=1 ttl=64 time=0.276 ms
64 bytes from 10.11.13.0: icmp_seq=2 ttl=64 time=0.045 ms

--- 10.11.13.0 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.045/0.160/0.276/0.115 ms
ns7@ubuntu:~/BSidesSLC-2021$ ^C
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns exec vnet10 ping -c2 10.11.12.0
PING 10.11.12.0 (10.11.12.0) 56(84) bytes of data.
64 bytes from 10.11.12.0: icmp_seq=1 ttl=64 time=0.055 ms
64 bytes from 10.11.12.0: icmp_seq=2 ttl=64 time=0.041 ms

--- 10.11.12.0 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1028ms
rtt min/avg/max/mdev = 0.041/0.048/0.055/0.007 ms
ns7@ubuntu:~/BSidesSLC-2021$
```

You can notice how we can now ping between the subnets!

From within Terminal 2: This is our 'server' network namespace:

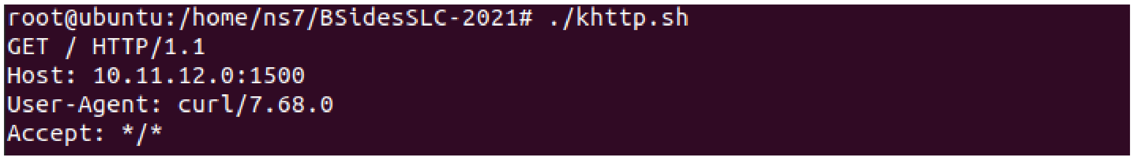
1. Start a bash process in the **vnet0** network namespace:
   a. sudo ip netns exec vnet0 bash
   b. ip a



```
ns7@ubuntu:~/BSidesSLC-2021$ sudo ip netns exec vnet0 bash
root@ubuntu:/home/ns7/BSidesSLC-2021# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: veth0@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default qlen 1000
    link/ether 86:50:3e:a2:2f:85 brd ff:ff:ff:ff:ff:ff link-netns vnet10
    inet 10.11.12.0/24 scope global veth0
       valid_lft forever preferred_lft forever
    inet6 fe80::8450:3eff:fea2:2f85/64 scope link
       valid_lft forever preferred_lft forever
5: veth11@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
 group default qlen 1000
    link/ether ea:94:1b:78:5e:d0 brd ff:ff:ff:ff:ff:ff link-netns vnet10
    inet6 fe80::e894:1bff:fe78:5ed0/64 scope link
       valid_lft forever preferred_lft forever
root@ubuntu:/home/ns7/BSidesSLC-2021#
```

   The output of **ip a** should show both veth interfaces assigned to vnet0 (veth0, veth11). Reminder: vnet0 is the network namespace that we're using for our 'server'

2. Start our 'kludgy http server' named `khttp.sh` which should be in the lab directory
   a. Lab directory: /home/class/Desktop/labs/network2
   b. Run ./khttp.sh



```
root@ubuntu:/home/ns7/BSidesSLC-2021# ./khttp.sh
GET / HTTP/1.1
Host: 10.11.12.0:1500
User-Agent: curl/7.68.0
Accept: */*
```

   Script ref:

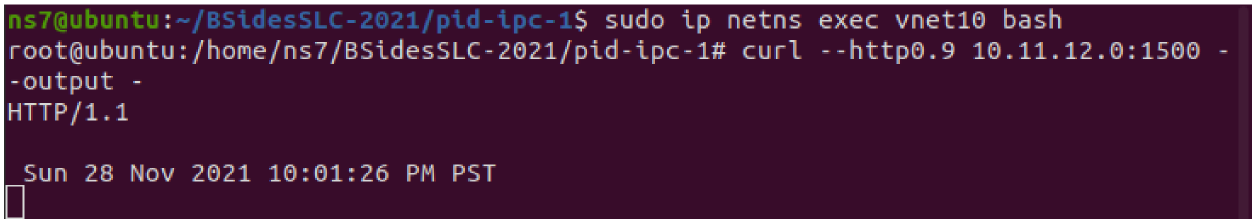
```
#!/bin/bash
#
# khttp - A kludgy http server using netcat. Please don't use this
#         for anything you care about

while true ; do (dd if=/dev/zero count=10000;echo -e "HTTP/1.1\n\n $(date)") | nc
-l  1500  ; done
```

** This script will act as a crude http server that can serve up HTTP0.9 via netcat

From within Terminal 3: This is our 'client' network namespace:

1. Enter the network namespace:
    a. sudo ip netns exec vnet10 bash

2. Try to reach the server sitting in vnet0 using curl:
    a. curl --http0.9 10.11.12.0:1500 --output -



```
ns7@ubuntu:~/BSidesSLC-2021/pid-ipc-1$ sudo ip netns exec vnet10 bash
root@ubuntu:/home/ns7/BSidesSLC-2021/pid-ipc-1# curl --http0.9 10.11.12.0:1500 -
-output -
HTTP/1.1

 Sun 28 Nov 2021 10:01:26 PM PST
```

Success! We get a response back which shows the date and time

Congratulations! You've successfully configured 2 network namespaces and associated network routing configurations which allow the namespaces to talk to each other

Bonus lab 1: Configure the 'client' namespace to be able to route to the public internet!
Hint: Check the route with **sudo ip -n vnet10 route**. Is there a path to 0.0.0.0/0?

Bonus lab 2: Try running **netstat -plnt** on the host and in each network namespace we created. What is listening where?
Hint: If it works correctly you should see a difference in output
Hint 2: You can use ip netns exec multiple times per-namespace

CGroups Lab 1

CGroups are the Linux method for controlling system resource usage. In this lab we will combine the namespace isolation from before with PID count limits and a deliberately malicious go process, to see what happens.
Note that both labs interact with CGroups V1 only. There is a V2 of Cgroups, but Systemd makes heavy use of V1 CGroups and it takes multiple configurations and some restarts to reconfigure it.

This lab takes you on a limited tour of CGroups. The topic is vast and this is just a taste to give you a sense of what this isolation mechanism is. Hopefully it spurs additional investigation if you're hungry for more :)

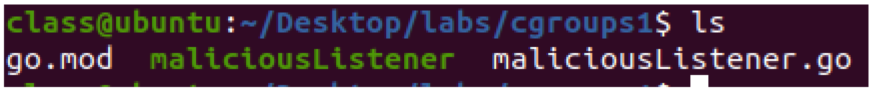
1. **Open two Terminal windows / tabs / instances / loci**

2. **Left Terminal: cd ~/Desktop/labs/cgroups1**

3. **In this directory you will find the program "maliciousListener.go"**
    a. This is a simple Go webserver that listens on port 8081 for GET requests
    b. The ?exec= request param accepts a single argument which is immediately executed, and the results are echoed back to you
    c. This sort-of imitates an evil command and control listener or deliberately vulnerable web application

4. **Left Terminal: go mod init cgroups1/maliciousListener**
    a. If it 'already exists', go to the next step

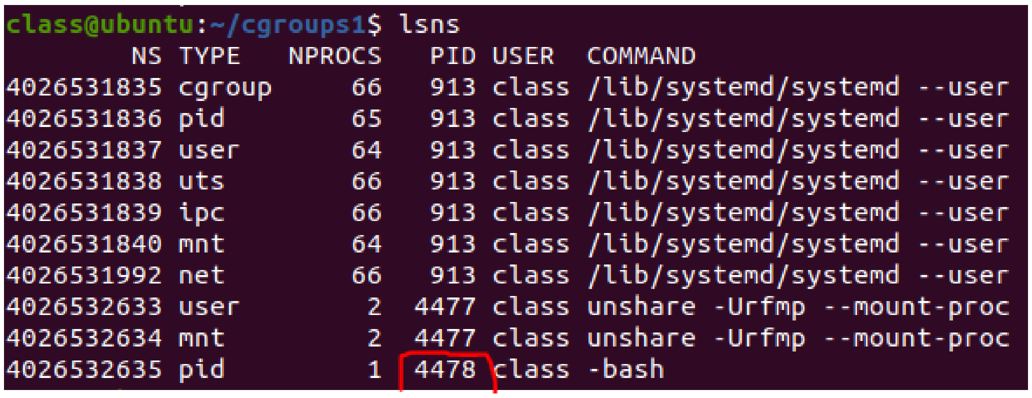
5. **Left Terminal: go build**
    a. You'll find the "maliciousListener" program in the current folder



```
class@ubuntu:~/Desktop/labs/cgroups1$ ls
go.mod  maliciousListener  maliciousListener.go
```

6. **Left Terminal: unshare -Urfmp --mount-proc**
    a. For simplicity's sake we're not going to bother with isolating away the primary / directory
    b. This command creates new user, mount, and pid namespaces, sets up your user as root inside the user namespace (again, without actual root powers), and mounts your namespace-isolated /proc over the original /proc point.
        i. To confirm this, you could run **"ps -ef"** and should only see a small number of processes running (PIDs likely 1 and 9-ish)
    c. Note that as before, you can still see the contents of the folder you were in beforehand.
        i. **"ls"** will report go.mod, maliciousListener, and maliciousListener.go

7. **Right Terminal (not where unshare was run): lsns**
    a. Keep track of the PID reported for the '-bash' process, shown here:

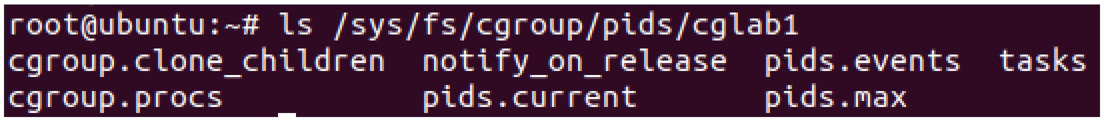


```
class@ubuntu:~/cgroups1$ lsns
        NS TYPE   NPROCS   PID USER  COMMAND
4026531835 cgroup     66   913 class /lib/systemd/systemd --user
4026531836 pid        65   913 class /lib/systemd/systemd --user
4026531837 user       64   913 class /lib/systemd/systemd --user
4026531838 uts        66   913 class /lib/systemd/systemd --user
4026531839 ipc        66   913 class /lib/systemd/systemd --user
4026531840 mnt        64   913 class /lib/systemd/systemd --user
4026531992 net        66   913 class /lib/systemd/systemd --user
4026532633 user        2  4477 class unshare -Urfmp --mount-proc
4026532634 mnt         2  4477 class unshare -Urfmp --mount-proc
4026532635 pid         1  4478 class -bash
```

8. **Right Terminal: sudo su -**
   a. Password is ‘classy’
   b. The PIDs cgroup must be administered by root

9. **Right Terminal: mkdir -p /sys/fs/cgroup/pids/cglab1**
   a. CGroups are administered like a filesystem
   b. This command is creating a new pids type CGroup called cglab1

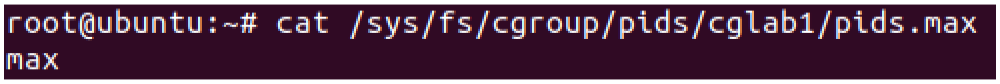
10. **Right Terminal: ls /sys/fs/cgroup/pids/cglab1**



```
root@ubuntu:~# ls /sys/fs/cgroup/pids/cglab1
cgroup.clone_children  notify_on_release  pids.events  tasks
cgroup.procs           pids.current       pids.max
```

   a. In here you see the files which are used to control the CGroup
   b. For pids, the primary files are cgroup.procs, pids.current, and pids.max

11. **Right Terminal: cat /sys/fs/cgroup/pids/cglab1/pids.max**



```
root@ubuntu:~# cat /sys/fs/cgroup/pids/cglab1/pids.max
max
```

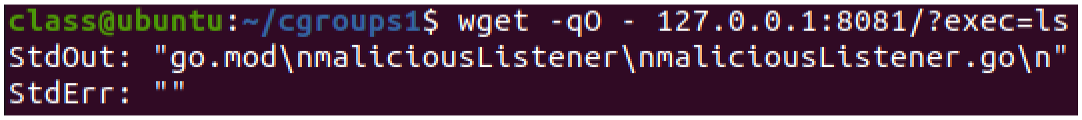
   a. Pids.max, as you might guess, is used to control the max number of pids which can exist across all processes in total in this CGroup.
   b. By default, it’s unlimited (that’s what ‘max’ mean)
   c. Let’s assign our PID namespace into this CGroup
   d. Find your pid namespace’s bash process PID from step 8

12. **Right Terminal: echo 4478 > /sys/fs/cgroup/pids/cglab1/cgroup.procs**
   a. Replace 4478 with your PID in the above step 8 command results
   b. This places your process into the CGroup, and can be verified with:
   c. cat /sys/fs/cgroup/pids/cglab1/cgroup.procs
      i. The response will be your process number (4478 in my case)

13. **Left Terminal: ./maliciousListener**
   a. This starts our little Go listener

b. Note that this process **must be started after placing the bash PID into the CGroup**
c. If the listener process is started first, then it will have its own PID. You might think it's a child of bash and so the CGroup will recursively apply to all children automatically - but this is not the case.
d. CGroups follow forks from a process, but do not automatically apply to child PIDs

14. Right Terminal: wget -qO - 127.0.0.1:8081/?exec=ls



```
class@ubuntu:~/cgroups1$ wget -qO - 127.0.0.1:8081/?exec=ls
StdOut: "go.mod\nmaliciousListener\nmaliciousListener.go\n"
StdErr: ""
```

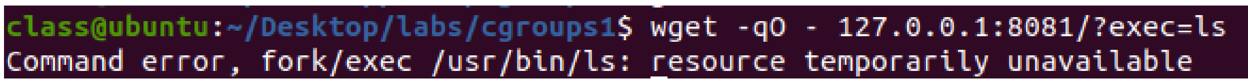
a. You see here that your 'ls' command ran without a hitch and told you about the directory from which the maliciousListener is running
b. Currently the CGroup is unrestricted and providing no process count protection

15. Right Terminal: cat /sys/fs/cgroup/pids/cglab1/cgroup.procs
a. Note that there are two entries! We only added one PID.
b. **CGroups follow process forks**, so when bash forked to execute the maliciousListener program it was inherited into the CGroup. This is logical, otherwise a fork is all it would take to evade CGroup limitations.

16. Right Terminal: cat /sys/fs/cgroup/pids/cglab1/pids.current
a. Your console will echo whatever the total PID count is in your namespace (this varies), in my case it was **7**
   i. This count can vary a bit, 6 or 7 is usually what's reported
b. Use the number which is returned here in the next step

17. Right Terminal: echo 6 > /sys/fs/cgroup/pids/cglab1/pids.max
a. This configures your CGroup to allow a maximum of N PIDs in total
b. Note that since you're setting the max pids to the number which already exists, there can be no additional pids in this CGroup
c. This means no additional processes can be run
   i. An important note here is that this refers specifically to **processes** and not **threads**. Threads have a separate ID system which is not controlled by the pids CGroup.

18. Right Terminal: wget -qO - 127.0.0.1:8081/?exec=ls
a. The result should be this error



```
class@ubuntu:~/Desktop/labs/cgroups1$ wget -qO - 127.0.0.1:8081/?exec=ls
Command error, fork/exec /usr/bin/ls: resource temporarily unavailable
```

There you have it. You configured the pids CGroup such that even though your process was literally just sitting there waiting to run some arbitrary command, it could do nothing because there were no additional PIDs available on fork.

This may be a contrived case, as the number of PIDs is not easy to track down and enforce. But this is a powerful control for when it can be done.

Considering a containerized environment, this can prevent a single container from fork-bombing its way to take down the entire host. CGroups also allows for memory and CPU cycle resource allocation which can further reduce the blast-radius associated with memory leaks and computation-intensive operations on shared underlying hosts.

CGroups Lab 2

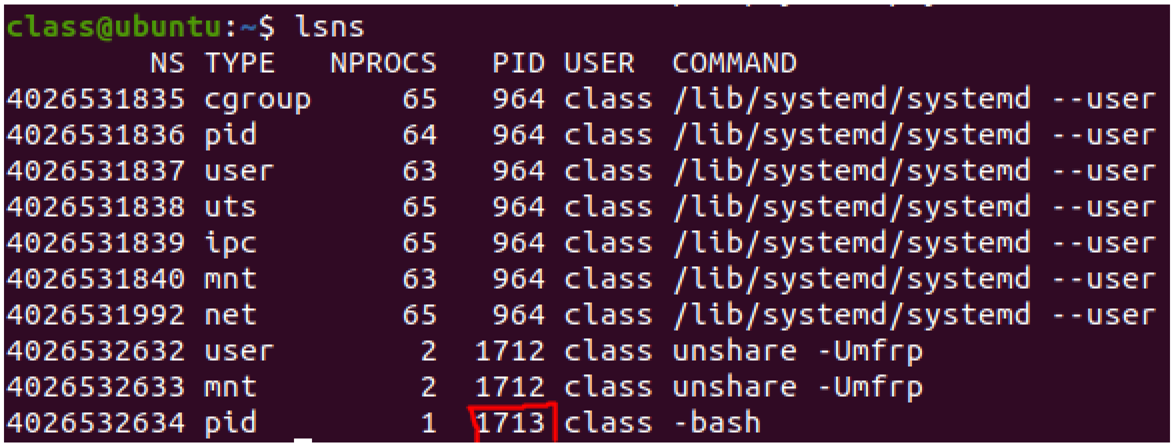
In this lab we learn about constraining CPU cycle usage via the cpu CGroup. This CGroup uses a divisor system to distribute resources to control groups.

cpu.cfs_period_us represents a time segment in microseconds (min allowed is 1 millisecond and max allowed is one full second). This represents one CPU core worth of time.
cpu.cfs_quota represents how much time during that segment may be used for processes in this CGroup.
If your period is 100000 (default, 100ms) and your quota is 50000 (50ms) then your process will run for 50% of the time available on a single CPU, for 50% usage of that CPU core. Given the same period, setting the quota to 150000 would allow 1.5 CPUs worth of usage.

1. **A Tale of Two Terminals**
    a. Yeah, you need two again

2. **Left Terminal: unshare -Umfrp**

3. **Right Terminal: lsns**
    a. We again need to record the bash process ID from the output

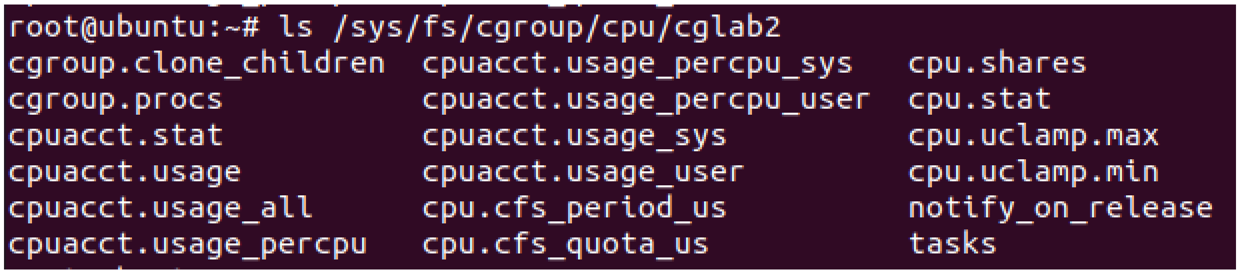


```
class@ubuntu:~$ lsns
        NS TYPE   NPROCS   PID USER  COMMAND
4026531835 cgroup     65   964 class /lib/systemd/systemd --user
4026531836 pid        64   964 class /lib/systemd/systemd --user
4026531837 user       63   964 class /lib/systemd/systemd --user
4026531838 uts        65   964 class /lib/systemd/systemd --user
4026531839 ipc        65   964 class /lib/systemd/systemd --user
4026531840 mnt        63   964 class /lib/systemd/systemd --user
4026531992 net        65   964 class /lib/systemd/systemd --user
4026532632 user        2  1712 class unshare -Umfrp
4026532633 mnt         2  1712 class unshare -Umfrp
4026532634 pid         1  1713 class -bash
```

4. **Right Terminal: sudo su -**
    a. Password is ‘classy’

5. **Right Terminal: mkdir /sys/fs/cgroup/cpu/cglab2**
    a. This creates the new cpu CGroup ‘cglab2’

6. **Right Terminal: ls /sys/fs/cgroup/cpu/cglab2**
    a. There are a lot of files in here



```
root@ubuntu:~# ls /sys/fs/cgroup/cpu/cglab2
cgroup.clone_children  cpuacct.usage_percpu_sys   cpu.shares
cgroup.procs           cpuacct.usage_percpu_user  cpu.stat
cpuacct.stat           cpuacct.usage_sys          cpu.uclamp.max
cpuacct.usage          cpuacct.usage_user         cpu.uclamp.min
cpuacct.usage_all      cpu.cfs_period_us          notify_on_release
cpuacct.usage_percpu   cpu.cfs_quota_us           tasks
```

    b. Note that this directory is shared between **cpuacct** and **cpu**. cpuacct is for ‘CPU Accounting’ and these files can be read to see usage, statistics, system CPU usage vs user, and so on.

7. **Right Terminal: cat /sys/fs/cgroup/cpu/cglab2/cpu.cfs_period_us**
    a. Default is 100 ms CPU time period
        i. 100000 is 100k microseconds
        ii. Why 1ms is the minimum but you configure it in microseconds I don’t really know

8. **Right Terminal: cat /sys/fs/cgroup/cpu/cglab2/cpu.cfs_quota_us**
    a. The -1 which you get back means ‘unrestricted’

9. **Now we place the PID namespace isolated bash process in this new cpu CGroup**
    a. Right Terminal: echo 1713 > /sys/fs/cgroup/cpu/cglab2/cgroup.procs
        i. You’ll have to change this PID to whatever lsns returned from step 3

10. Right Terminal: echo 50000 > /sys/fs/cgroup/cpu/cglab2/cpu.cfs_quota_us

a. Sets a limit of 50ms of execution per CPU execution period
b. The execution period is 100ms, so this is a 50% limit on a single CPU

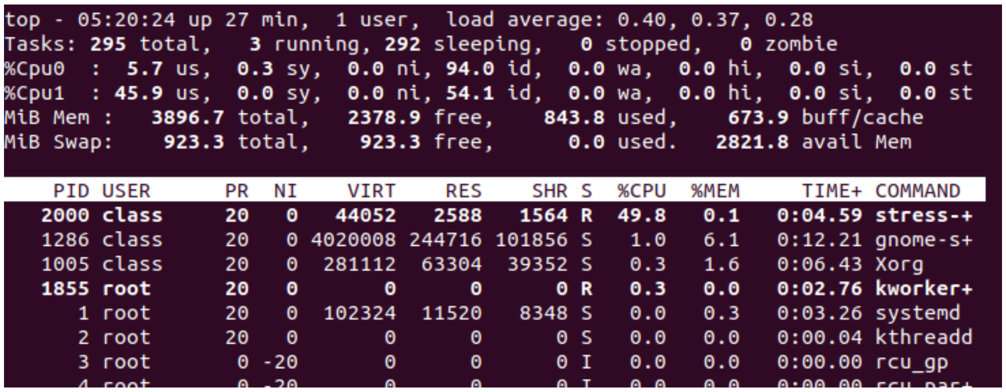
11. Let's run something CPU intensive and see how the system throttles it

12. Right Terminal: top

a. While top is running, press the number 1 and you get stats per CPU

13. Left Terminal: stress-ng --cpu 1 --cpu-method matrixprod --metrics-brief --perf -t 60

a. For this run of stress-ng we're using a single CPU core and the results are pretty much what you would expect. Run **top** over in the **Right Terminal**:

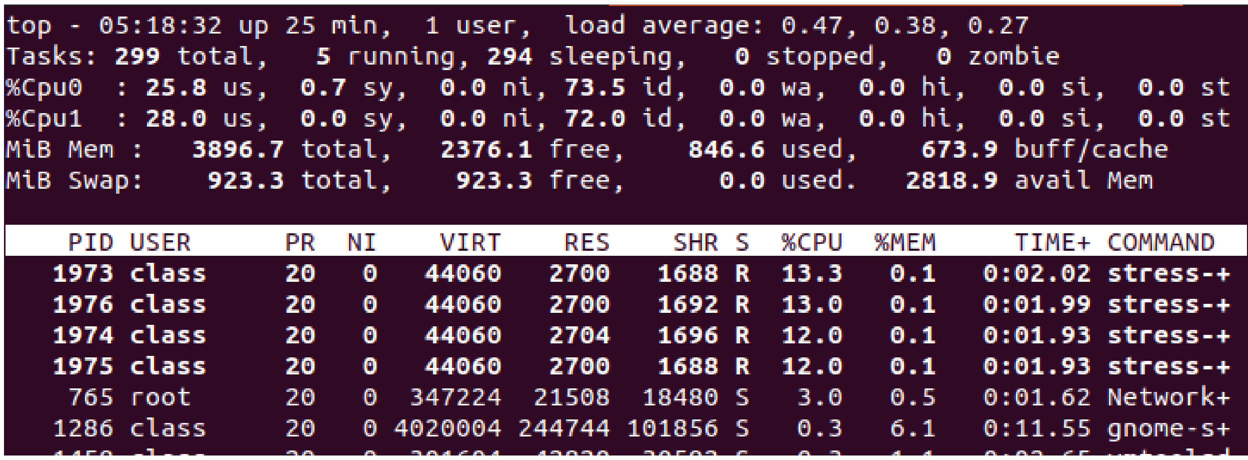


```
top - 05:20:24 up 27 min,  1 user,  load average: 0.40, 0.37, 0.28
Tasks: 295 total,   3 running, 292 sleeping,   0 stopped,   0 zombie
%Cpu0  :  5.7 us,  0.3 sy,  0.0 ni, 94.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 45.9 us,  0.0 sy,  0.0 ni, 54.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3896.7 total,   2378.9 free,    843.8 used,    673.9 buff/cache
MiB Swap:    923.3 total,    923.3 free,      0.0 used.   2821.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2000 class     20   0   44052   2588   1564 R  49.8   0.1   0:04.59 stress-+
   1286 class     20   0 4020008 244716 101856 S   1.0   6.1   0:12.21 gnome-s+
   1005 class     20   0  281112  63304  39352 S   0.3   1.6   0:06.43 Xorg
   1855 root      20   0       0      0      0 R   0.3   0.0   0:02.76 kworker+
      1 root      20   0  102324  11520   8348 S   0.0   0.3   0:03.26 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.04 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
```

b. We configured the cpu CGroup with 50% of the available quota for one CPU at runtime, and there it is - right around 50% of one CPU is used.
c. Let's make it interesting and re-run the stress test specifying multiple CPUs

14. Left Terminal: stress-ng --cpu 4 --cpu-method matrixprod --metrics-brief --perf -t 60

a. How do the **top** statistics look over in the **Right Terminal**?



```
top - 05:18:32 up 25 min,  1 user,  load average: 0.47, 0.38, 0.27
Tasks: 299 total,   5 running, 294 sleeping,   0 stopped,   0 zombie
%Cpu0  : 25.8 us,  0.7 sy,  0.0 ni, 73.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 28.0 us,  0.0 sy,  0.0 ni, 72.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3896.7 total,   2376.1 free,    846.6 used,    673.9 buff/cache
MiB Swap:    923.3 total,    923.3 free,      0.0 used.   2818.9 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1973 class     20   0   44060   2700   1688 R  13.3   0.1   0:02.02 stress-+
   1976 class     20   0   44060   2700   1692 R  13.0   0.1   0:01.99 stress-+
   1974 class     20   0   44060   2704   1696 R  12.0   0.1   0:01.93 stress-+
   1975 class     20   0   44060   2700   1688 R  12.0   0.1   0:01.93 stress-+
    765 root      20   0  347224  21508  18480 S   3.0   0.5   0:01.62 Network+
   1286 class     20   0 4020004 244744 101856 S   0.3   6.1   0:11.55 gnome-s+
```

b. At first glance you might think "What? Each CPU is at 25% instead of 50%."

c. The quota we set in the cpu CGroup represents usage of a single CPU. 50% of a single CPU is effectively the same as 25% of two CPUs.
d. If your VM had 8 CPUs and this command was re-run with **--cpu 8** then each one would only see about 6.25% usage.
e. Also of interest here is that each stress-ng process listed by **top** (four of them) is getting half of the available CPU quota. My VM only has two CPUs, but we're running four processes. The CPU scheduler tries to give equal time to all processes which are placed in a given CPU CGroup.

As additional exercises, try changing the CPU quota in the CGroup up and down. Re-run the stress test each time with differing **--cpu** counts and see what happens.

Seccomp-bpf Lab 1

Seccomp-bpf allows you to control which syscalls are allowed to be used by your application process. For this to work without crashing your application will need to be profiled to identify which syscalls are 'normal' and expected.

These labs use golang, `github.com/seccomp/libseccomp-golang` and libseccomp-dev, all of which are supplied in the VM

Lab 1 shows you how to profile a mock-application, which serves to illustrate the concepts involved. It should be noted that production applications are substantially more complex and that additional, rigorous testing should be employed before applying seccomp filters to production infrastructure.

Steps

1. Create a simple application. This program should exist in your lab directory as **seccomp1.go**:

```go
package main

import (
  "log"
  "syscall"
  "time"
)

func main() {
  dt := time.Now().Format("2006-01-02T15-04-05")
  errMkdir := syscall.Mkdir("tmpdir-"+dt, 0644)
  if errMkdir != nil{
     panic(errMkdir)
  }

  log.Println("Created tmpdir via syscall")
}
```

2. Compile the application. You should see an executable, in this case named **'seccomp1'**:

```
$ go build -o seccomp1
$
$ ls -la
total 1868
drwxrwxr-x 4 ns7 ns7    4096 Nov 27 16:08 ./
drwxrwxr-x 5 ns7 ns7    4096 Nov 27 15:50 ../
-rw-rw-r-- 1 rion ns7      40 Nov 27 15:50 go.mod
-rwxrwxr-x 1 ns7 ns7 1884755 Nov 27 16:08 seccomp1*
-rw-rw-r-- 1 ns7 ns7     249 Nov 27 16:05 seccomp1.go
$
```

3. Use strace to profile the application execution. The output will list the syscalls used by the application:

```
$ strace -c ./seccomp1
2021/11/27 16:08:17 Created tmpdir via syscall
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 57.18    0.000506           4       113           rt_sigaction
 15.59    0.000138          46         3           clone
```

```
  9.04    0.000080         80         1          mkdirat
  5.88    0.000052          6         8         rt_sigprocmask
  2.26    0.000020         10          2          rt_sigreturn
  2.26    0.000020         10          2          futex
  1.81    0.000016          5         3         fcntl
  1.58    0.000014          7         2         openat
  1.47    0.000013          4         3         read
  1.24    0.000011         11          1          write
  0.90    0.000008          0        19          mmap
  0.79    0.000007          3         2         close
  0.00    0.000000          0         1         execve
  0.00    0.000000          0         2         sigaltstack
  0.00    0.000000          0         1         arch_prctl
  0.00    0.000000          0         1         gettid
  0.00    0.000000          0         1         sched_getaffinity
------ ----------- ----------- --------- --------- ----------------
100.00    0.000885              165          total
```

4. The syscalls will be used in the next lab. You'll note a random directory was created in the working directory where the executable was run. This is the expected output:

   Syscalls: rt_sigaction, clone, mkdirat, rt_sigprocmask, rt_sigreturn, futex, fcntl, openat, read, write, mmap, close, execve, sigaltstack, arch_prctl, gettid, sched_getaffinity, **exit_group**

   Note: exit_group should be appended to the list of approved syscalls otherwise you get into a messy situation where your program never quits! I've added it to the list above even though it doesn't show up in the strace for completeness

Seccomp-bpf Lab 2

Lab 2 shows how an application can be written in a way that implements seccomp to limit the syscalls that can be called by the application process. Without a shim of some kind a developer would need to implement this in-program as we demonstrate here

At the end of this lab you will have a program that is capable of configuring seccomp policies. Additionally, you will discover how a program behaves when a necessary syscall is omitted from the seccomp whitelist

1. Create a simple application. This program should exist in your lab directory as **seccomp2.go**:

```
package main

import (
  "fmt"
  libseccomp "github.com/seccomp/libseccomp-golang"
  "log"
  "syscall"
  "time"
)

func allowSyscalls(syscalls []string){
  // Setup the 'default deny' policy for syscalls not in the list
  returnCode := int16(syscall.EPERM)
  seccompAction := libseccomp.ActErrno.SetReturnCode(returnCode)
  filter, filterErr := libseccomp.NewFilter(seccompAction)
  if filterErr != nil {
     log.Println("Unable to create seccomp filter, Err:", filterErr)
  }
  fmt.Println("Configured default-deny for any non-approved syscalls")

  // Load the approved list of syscalls
  for _, syscallN := range syscalls{
     fmt.Println("Allowing ", syscallN)
     scmpSyscall, scmpSyscallErr := libseccomp.GetSyscallFromName(syscallN)
     if scmpSyscallErr != nil{
        panic(scmpSyscallErr)
     }

  filter.AddRule(scmpSyscall, libseccomp.ActAllow)
  }

  filterLoadErr := filter.Load()
  if filterLoadErr != nil{
     log.Println("Unable to load syscall filter! Err:", filterLoadErr)
  }
}

func main() {
  allowedSyscalls := []string{"rt_sigaction", "mkdirat", "clone",
"rt_sigprocmask", "rt_sigreturn", "futex", "fcntl",
     "openat", "read", "write", "mmap", "close", "execve", "sigaltstack",
"arch_prctl", "gettid",
     "sched_getaffinity", "exit_group"}
  allowSyscalls(allowedSyscalls)

  dt := time.Now().Format("2006-01-02T15-04-05")
  errMkdir := syscall.Mkdir("tmpdir-"+dt, 0644)
  if errMkdir != nil{
     log.Println("Unable to create directory via syscall!")
     panic(errMkdir)
  }

  log.Println("Created tmpdir via syscall")
}
```

2. Compile the application. You should see an executable, in this case named **'seccomp2'**:

```
$ go build -o seccomp2
$
$ ls -la
total 5760
drwxrwxr-x 4 ns7 ns7    4096 Nov 27 16:08 ./
drwxrwxr-x 5 ns7 ns7    4096 Nov 27 15:50 ../
-rw-rw-r-- 1 rion ns7      40 Nov 27 15:50 go.mod
-rwxrwxr-x 1 ns7 ns7 1989664 Nov 27 19:36 seccomp2*
-rw-rw-r-- 1 ns7 ns7     1294 Nov 27 19:30 seccomp2.go
$
```

3. Execute the application and note the output

```
$ ./seccomp2
Configured default-deny for any non-approved syscalls
Allowing  rt_sigaction
Allowing  clone
Allowing  mkdirat
Allowing  rt_sigprocmask
Allowing  rt_sigreturn
Allowing  futex
Allowing  fcntl
Allowing  openat
Allowing  read
Allowing  write
Allowing  mmap
Allowing  close
Allowing  execve
Allowing  sigaltstack
Allowing  arch_prctl
Allowing  gettid
Allowing  sched_getaffinity
Allowing exit_group
2021/11/27 19:40:35 Created tmpdir via syscall

$ ls -la
total 5764
drwxrwxr-x 5 ns7 ns7    4096 Nov 27 19:40 .
drwxrwxr-x 5 ns7 ns7    4096 Nov 27 15:50 ..
-rw-rw-r-- 1 ns7 ns7      93 Nov 27 19:24 go.mod
-rw-rw-r-- 1 ns7 ns7     191 Nov 27 19:24 go.sum
-rwxrwxr-x 1 ns7 ns7 1989664 Nov 27 19:36 seccomp2
drw-r--r-- 2 ns7 ns7    4096 Nov 27 19:40 tmpdir-2021-11-27T19-40-35
```

You'll note that the program allowed a number of syscalls, then proceeded to generate a new randomly generated directory name in the working directory where the application executed from

4. Now… let's see what happens when we **remove a required syscall** from the approved list. To do this, remove `, "mkdirat"` from the list of syscalls on **line 40 of seccomp2.go**

5. Re-run the build command with a different output filename:

```
$ go build -o seccomp2.bad
```

6. Run the 'bad' executable and note the output:

```
$2./seccomp2.bad

…

021/11/27 20:37:21 Unable to create directory via syscall!
panic: operation not permitted

goroutine 1 [running]:
```

```
main.main()
	/home/ns7/bsidesslc2021-seccomp-2/seccomp2.go:49 +0x245
```

You'll note in the output that the panic indicates **operation not permitted**! This shows that the syscall was successfully blocked and that the default action we set for unconfigured syscalls was leveraged (**EPERM**, or 'error permanent')

Congratulations on making it through lab 2! It's worth noting that seccomp-bpf is easy to retrofit without code-changes using containerd or docker. We go through the exercise here to show how it works 'without docker'.

Follow-on to this lab: Review the docker default seccomp profile (https://github.com/moby/moby/blob/master/profiles/seccomp/default.json) to understand what syscalls are allowed/disallowed.

As a bonus-exercise, try crafting your own seccomp profile json that can be used in docker!