TD 5: Virtualization, micro-kernels

In this practice lesson, we will discuss a few questions for the various virtualization technologies:

Guest modification: does the technology require modifying the source code of the guest?

Isolation: to what extent does the technology isolate guests?

Execution speed: is the guest slowed down by the technology?

1 Environment

- Restart your test VM based on Debian 11. As a reminder, the login/pass is cremi/cremi.
- Install the `debootstrap` package.
- Run

  `sudo debootstrap stable /var/tmp/chroot`

  and let it proceed while you continue the steps below.
- Install the `docker.io` package.
- Run

  `sudo addgroup cremi docker`

  to let your user use docker.
- Run

  `newgrp docker`

  to run a shell with the new docker group permission.
- Run `wget https://dept-info.labri.fr/~thibault/SecuSys/Dockerfile`
- In that shell, run

  `docker build -t mydocker .`

  and let it proceed while you continue the steps below. Notice that it is installing packages etc. while we have seemingly run `docker` as normal user.

- On your **host machine**, `cd` to `/local/yourlogin` (mkdir it if it doesn't exist yet) and run[1]

  `tar xfS ~sathibau/debian-hurd.img.tar.xz`

  and let it proceed while you continue the steps below.

- Once the commands in your VM are finished, log out and log in again inside the VM (so that all your shells in your VM have docker group permission without having to care)

2 Unix process

You are used to Unix processes, but possibly never realized that they are actually running inside a virtualized environment: a virtual addressing space. The program can do whatever it wants inside it, and does not have any access outside it.

- To what extent is your program source code "modified" compared to if it was running on bare metal, without the notion of process?
- To what extent is your program running in the process isolated from other processes?
- To what extend is your program running as fast as without this virtualization technique (in terms of CPU speed, and devices speed).

3 chroots

Chroots is a very simple container technology. Enter and configure the chroot with this:

- Run `sudo chroot /var/tmp/chroot`
- Run `mount none /proc -t proc`
- Run `mount none /sys -t sysfs`
- Run `mount udev /dev -t devtmpfs`
- Look around in the directories, you will notably see that /home is empty. Try to create a file somewhere. See in another terminal that the file appears inside `/var/tmp/chroot/`

  `chroot` indeed means that you are stuck inside the chroot directory, i.e. `/` is stuck at `/var/tmp/chroot` (CHange ROOT).
- Install the `pciutils` package inside the chroot.
- Run `dmesg`, `ip a ls`, `lspci`, `lsmod`
- Compare with the same commands in your Debian11 VM (run with `sudo`)
- In another terminal (thus outside the chroot), run `sudo modprobe ledtrig_timer`

[1] That's coming from `http://cdimage.debian.org/cdimage/ports/latest/hurd-i386/debian-hurd.img.tar.xz`

- See that inside the chroot the module showed up in `lsmod`
- Inside the chroot, run `rmmod ledtrig_timer`
- In another terminal, see that the module disappeared from `lsmod`
- Run `ps axu | grep bash` both inside the chroot and in another terminal, compare the outputs.

So programs running inside the chroot are basically seeing all of their host environment: network configuration, kernel modules, processes, etc. And "root" inside the chroot really has all root powers... It can for instance open the disk /dev/sda...

Worse, it's trivial to escape the chroot when running as root:

- Inside the chroot, install `gcc`
- Create a C program that does the following:

```
mkdir("tmp", 0755);
chroot("tmp");
chroot("../../../../..");
execl("/bin/sh", "-i", NULL);
```

- Compile it, run it, run `ls /home`, you're back to the host!

Chroots are really not a secure virtualization technique. It was never meant to be, actually.

- Answer again the three "to what extent" questions at the end of the previous section.

(Exit from your chroot)

4 docker

- Run

  ```
  docker run -t -i mydocker
  ```

- In another terminal, run `mount | grep docker`
- See the overlay directory (`/var/lib/docker/...`). Try to create a file inside the docker shell, see it appear in the overlay directory.
- Try the same commands as in the previous section. See that most of them fail out of permissions
- Compare the outputs of `ip a ls` inside the docker shell and outside. Notice the `docker0` interface outside docker, it's bound to the `eth0@if11` interface inside docker so that the docker shell has network access. Notice their IP addresses.
- Also compare the outputs of `ps axu | grep bash`

- Run `df -h .` both inside and outside. You will see that the docker root is an overlay between the host's filesystem and the docker shell.
- You can check that capabilities you actually have with `capsh --print`
- Compare the output of that command from inside the docker shell and outside it (but as root with `sudo`). The interesting part is the `Current` line. `=ep` means all permissions are permitted and actually effective. (More on the capabilities in `man 7 capabilities`)
- Notice that there is the `cap_mknod` capability. That's supposed to let us create arbitrary device access...
- Try to `mknod /dev/sda b 8 0`
- Try to run `fdisk -l /dev/sda`
- No, that doesn't work...
- The `chroot("../../../../..")` trick above won't work either.

So docker images are much more meant to be secure.

Note that there is a `--privileged` flag which allows more inside the docker shell, but that's way too much and there are various exploits that can subvert it.

- Answer again the three "to what extent" questions.

5 VM

Let's get back to our Debian11 VM

- You have already played with `dmesg`, `ip`, `lsmod` etc. in that VM, you know very well that it's completely unrelated to the real system.
- Just to be sure, run `ps axu | grep bash` both inside and outside the VM, compare the outputs.
- Outside the VM, run `ps axu | grep Box`, notice the `VirtualBoxVM` process, and see that it consumed some CPU time. That's representing the VM in the host.

So this is really emulation: VirtualBox is simulating a complete PC with a virtual processor, etc. Let's look more.

- Both inside and outside, run `lspci`

All of what is showing up inside the VM is actually emulated hardware. This does not seem obvious because virtualbox is emulating well-known hardware so that the Operating System can work with its usual device drivers.

- Install the `hdparm` package

- Run `hdparm -i /dev/sda`

Ok, there we see that the disk model is labelled VBOX, it's indeed a disk simulated by VirtualBox.

- `cat /proc/cpuinfo`
- Compare the `flags` part. Notably notice the presence of the sse and avx flags.
- Compare the `bugs` part.

Since VirtualBox is using hardware CPU acceleration, the available instruction set is basically the same. There are however some parts which are disabled, to avoid a blue pill escape...

- Notice that inside the VM, the `vmx` (or `svm` on AMD64) is not available. This means we cannot have *nested* hardware acceleration (called EPT by Intel): if we run virtualbox/KVM inside that virtualbox VM, it will be very slow.
- If you have a USB device, you can tell virtualbox to perform *USB pass-through*, i.e. *give* the USB device to the VM, and it's the VM's driver which will drive it.
- Answer again the three "to what extent" questions.
- Note that it seems that the VirtualBox virtualization technique is considered to be safe, since it's allowed in Cremi. docker, however, is not (yet) :)

6 Web navigator Java Virtual Machine (JVM)

Let's consider a Javascript script, a Java applet or Flash plugin running inside the web navigator. Answer again the three "to what extent" questions.

7 (if you have time) GNU/Hurd

Let's now have a look at that debian-hurd image.

- Make sure that VirtualBox is closed (hardware acceleration cannot work with both VirtualBox and kvm running at the same time).
- `cd /local/yourlogin`
- Run `kvm -m 1G -snapshot -hda debian-hurd.img`
- Note: if at any time you don't see your mouse cursor any more, see the title of the window: press ctrl-alt-G to get out of the "grab" mode.
- Log in as `demo`
- Run `ps axu --width=80 | less` to see all the userland processes.

- Notably see the presence of `ext2fs`, `netdde`, `pfinet`. `ext2fs` is the filesystem translator, `netdde` is running the network driver, and `pfinet` implements the TCP/IP stack. The disk driver is still inside the kernel but there are plans to move it out to a userland process.
- Type `q` to quit `less`
- Run `wget www.gnu.org`
- Run `sudo killall netdde` to kill the network driver.
- Run `wget www.gnu.org` again, see that some messages appear about an irq handler cleanup: it's the netdde driver just starting again, and we can use the network again.
- `cd /ftp://ftp.gnu.org/pub/gnu/hurd/`
- Notice that the usual shell commands work there.
- Run `ps axu | grep ftpfs` and see there that an ftpfs process appeared. Why is it running as `nobody` rather than as `demo`?
- Run `ls -ld /ftp:` and `showtrans /ftp:`
- This means that it's a *translated* directory: its content is *served* by the hostmux translator which starts an ftpfs translator.
- Run `cd` to get back to `demo`'s home directory
- Run `settrans -cap mytest /hurd/ftpfs ftp.gnu.org`
- Look in the `mytest` directory
- See with the `ps` command above that new ftpfs process appeared under the `demo` identity.
- Run
  `settrans -cap ~/myiso /hurd/iso9660fs ~/mytest/old-gnu/gnu-f2/hurd-F2-main.iso`
- `cd ~/myiso`, and see that we transparently access the inside of the ISO image from the ftp server.
- Reboot the system with `sudo reboot`
- Notice that these directory are still set up so, it's recorded in the filesystem!
- Create an empty file with `dd < /dev/zero > image bs=1M count=10`
- Format it with `/sbin/mke2fs -E root_owner image`
- Mount it with `settrans -cap mydir /hurd/ext2fs image` (ignore the warning about shutdown notification, it just tells you that it risks not getting notified if the system shuts down)
- Look in the `mydir` directory, run `df .` inside it.

- Notice with the `ps` command above that a new `ext2fs` process appeared, under the `demo` identity. See the other processes, that correspond to everything we have started under the `demo` identity.

So by putting most of the implementation of the system outside of the kernel, this allowed to delegate a lot of power to users, notably to implement their own filesystems, in a safe way since they are running under their identity! That's just using the available CPU time, memory space, disk space, network access.

That's also safe for mounting a USB stick found in the street, at worse it'll hack only the user that started `ext2fs`, not the kernel.

Linux has a similarly-looking support with FUSE, but it's mostly never enabled on production systems, as considered not really safe.

- Make a drawing that lays out all the translators you have seen, and their relations

Funny things then become possible:

```
touch hello
cat hello
settrans hello /hurd/hello
cat hello
settrans -fg hello /hurd/run echo yay
cat hello
settrans -fg hello /hurd/run /usr/games/fortune
cat hello
cat hello
cat hello
```

So that's a file whose content changes on each open, and which is simply the output of the `fortune` command!