SECURITY & ISOLATION

Resource limits

- Any container has to execute some application/command to stay running. This creates one or more processes that consumes the hosts memory and CPU.
- If the amount of available resources (CPU and Memory) runs out, this will cause severe performance degradation for the process and it may eventually crash.
- Using up all the resources on the system will not only affect the process inside the container, but also other processes owned by other users on the system. This may cause drastic effects on the overall performance of the machine.
- For that reason, Docker provides some controls that you can use to limit the amount of resources used by a given container when it starts up. They are discussed as follows.

Memory allowance

- You can limit the amount of memory that can be used by the container by using the `–m` or `––memory` flag.
- For example, the following container can only consume up to 512 MB of memory:

```
docker run --name webserver –m 512m httpd
```

- The amount of memory can be specified in bytes (b), kilobytes (k), megabytes (m) or gigabytes (g).
- Notice that the amount of memory specified by –m is not a reservation. That is, Docker does not guarantee that the application will always have access to the specified amount of memory. It will just prevent the container from exceeding that amount.
- Before you can assign the maximum memory the container can consume, you will have to carefully plan it. You will have to determine whether or not the application can function properly within the assigned memory limits and also whether or not the host can avail that amount of memory.
- Sometimes it is hard to determine the memory requirements of open-source programs. For well known applications like database engines and webservers, the suitable amount of memory may be calculated by database or system administrators. This often ends up with assigning an overestimated amount and working with trials and errors until reaching the optimum values.
- If any application runs out of memory it may start writing error messages indicating that in the logs and it may crash. Docker cannot help mitigate those risks so the best precaution is to use the `––restart` discussed previously in this course

CPU allowance

- The same thing can be done with the CPU, which is also a limited resource.
- Running out of CPU has different consequences than running out of memory. While in the second the process may run slower then eventually crash, running out of CPU will make the process wait indefinitely until some cycles get free to serve it.
- Sometimes it's better that the process crashes and gets restarted automatically (like by Docker) than to keep waiting and freezing the thread.
- Docker lets you specify a CPU limit in one of two ways:
  - *By assigning a "relative weight" to the container. This is the percentage of CPU cycles the container will use relative to the sum of all computing cycles available to the other containers.*
  - *By restricting the container to work on only a set of CPU cores.*
- Let's look at both methods next

Assigning a percentage of cycles to the container

- In this mode, you assign a number of cycles to the container while creating or running it. This is done using the `--cpu-shares` as follows:
  ```
  docker run –d --cpu-shares 1024 –name hogger1
  afakharany/hogger
  docker run –d --cpu-shares 2048 –name hogger2
  afakharany/hogger
  ```
  The above commands create two containers, hogger1 and hogger2. They run a CPU intensive command called "stress". Assigning 1024 cycles to the first and 2048 to the second means that hogger1 will get half the amount of CPU cycles available to hogger2.
- As more and more containers are created, the number that follows --cpu-shares controls the percentage of CPU assigned to the container relative to other containers that are currently running.
- Notice that CPU limiting differs than memory limiting; if a container needs more CPU than the assigned to it and that amount of CPU cycles is available on the host, the container will just break the limit and use the extra cycles. However, if those cycles were originally assigned to another container, which demanded them back, they will be assigned to it. This ensures that all containers use the maximum possible CPU resoures.

Assigning a CPU set to the container

- The other method of limiting the CPU resource is to assign a group of CPU's to the container.
- Most computers (if not all of them) nowadays have more than one core. You can assign a number of CPUs on which the container will run using the –cpuset-cpus flag. For example:
  ```
  docker run –d --name hogger --cpuset-cpus 0
  afakharany/hogger –c 1
  ```
- It can be specified in the following ways:
  - *`--cpu-set 0`* *for using only the first CPU of the host*
  - *`--cpu-set 0-2`* *for using the first, the second, and the third CPUs*
  - *`--cpu-set 0,2`* *for using the first and the third processors*

Controlling access to devices

- This is different than CPU and memory allowances; you do not limit a container's access the host devices, but you control it.
- You can have all types of devices on a Linux system. Some of those are automatically mapped by Docker and some others aren't.
- To map a device from the host to the container you can use the --device flag. For example, the following command:
  ```
  docker run -it --rm --privileged --device /dev/sr0:/dev/sr0
  busybox mkdir /media && mount /dev/sr0 /media && ls -l
  /media
  ```
  will map the cdrom device on the host (/dev /sr0) to a device with the same name on the container. Notice that we used the --privileged flag to be able to mount the device.

Docker users

- By default, Docker containers run as the root use account. This behavior might cause a potential security threat; as all daemons and services will be running using the administrator's privileges. If there is a something wrong (like a bug) with the application running inside the container, the consequences may be disastrous.
- However, sometimes you may explicitly need the container to be running under the root account powers to perform some essential system administration tasks.
- Sometimes you may even need root/admin access to the host operating system to perform some security related tasks.
- In the following we are going to discuss how Docker deals with all this scenarios.

Determining the container's user account

- You may need to know which user account will be used inside a container, such information is not available on the image; as there is no way to examine the images metadata.
- Naturally, you will want to inspect the container for the user and UID it is going to be using before you start running it. You can use the docker inspect command, discussed earlier, for this task. For example:

  ```
  docker create –name myUser busybox sleep 3000
  docker inspect --format="{{ .Config.User }}" myUser
  ```
- If the result of the above command is nothing, then the default user is root with UID 0. Otherwise, the author of the image has chosen to make it "run-as" a specific user.
- But this method has two problems:
  - *The use can always be changed easily in any script that the container uses.*
  - *You have to create a container before you can acquire this information. That may cause potential threats.*
- One possible solution for the first problem is to run a command like the following:

  ```
  docker run --rm --entrypoint "" busybox:latest id
  ```

  The above command will run a quick test on the busybox image to determine the user account by which the container will be run. Notice that the entrypoint command has been omitted (defaults to sh –c as mentioned before) to ensure that the id command will be the only one to run on the container. The id command is used to display the UID and the username of the current user.

Changing the default user

- You can change the default user that is used to run the container when running or creating it.
- However, to do this you must select a user that already exists in the container. You can use the following command to list the available users on a give container:
  ```
  docker run --rm -it --entrypoint "" busybox cat /etc/passwd | cut -d
  : -f 1,3,4,5
  ```
- Once you identified the user that you want to use with the container, you can issue the following command to run the container with that user:
  ```
  docker run --user nobody busybox id
  ```
  As you can see, the id command reported that the current user is nobody with an id of 99 (the UID value may or may not differ depending on the image you are using).
- The --user (or –u) option can be used also for specifying groups, UIDs and GIDs. For example:
  ```
  docker run --user 99 busybox id
  ```
  will achieve the same task as the previous command (nobody as UID of 99)
  ```
  docker run --user www-data:www-data busybox id
  ```
  which will run the container with t e www-data (Apache) user and www-data group and so on.
- Notice that a user can very easily promote itself to the root account on some weak images like BusyBox, which has the root account without a password. Stronger images like Ubuntu has this account locked by default. Accordingly, you should develop the habit of setting passwords and/or locking the root account in containers that do not do that by default.

LAB - Handling permissions on volumes

- The most practical use to changing the user which is used by default inside the container is handling file and directory permissions specially when using volumes.
- Let’s consider the following example:

```
echo "file contents" > sharedfile.txt
chmod 600 sharedfile.txt
ls -l sharedfile.txt
-rw------- 1 root root 14 Aug  6 08:52 sharedfile.txt
docker run -it --rm -v /share busybox cat /share/sharedfile.txt
cat: can't open '/share/sharedfile.txt': No such file or directory
docker run -it --rm -v /share:/share busybox cat
/share/sharedfile.txt
file contents
docker run -it -u nobody --rm -v /share:/share busybox cat
/share/sharedfile.txt
cat: can't open '/share/sharedfile.txt': Permission denied
```

- In this lab we create a sample text file and set the permissions so that it is only accessible by root. Then we create a container with the default account (root), and and present the file location as a bind volume. Trying to read the file contents this way is successful; as both the root account on the container and the one on the host share the same UID 0. Then we repeat the last step but this time using the nobody user. The nobody user cannot access the text file because the UID is different.