Security Engineering Coursework – Guideline

1 Environment Setup using Container

In this coursework, we need three machines. We use containers to set up the environment, which is depicted in Figure 1. In this setup, we have an attacker machine (Host M), which is used to launch attacks against the other two machines, Host A and Host B. These three machines must be on the same LAN, because the ARP cache poisoning attack is limited to LAN. We use containers to set up the lab environment.

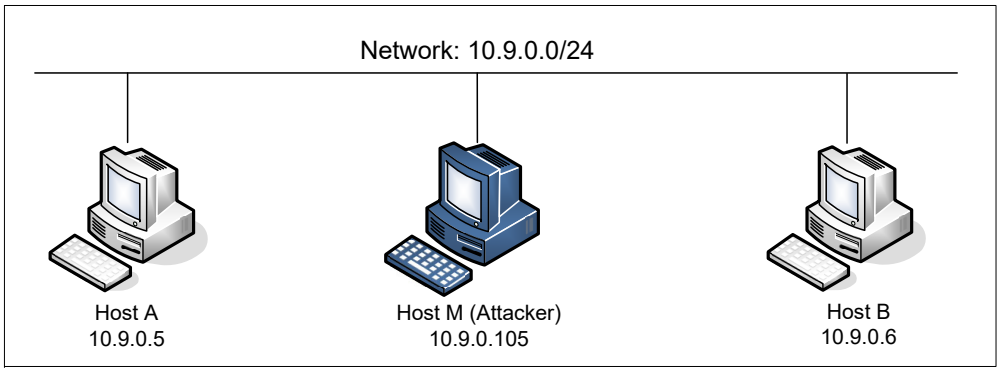

Figure 1: Lab environment setup

1.1 Container Setup and Commands

Please download the `Courseworksetup.zip` file to your VM from the Ultra Blackboard's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Ths VM is the same one as you already used in your at-home practicals. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is available in the "At-home practical" folder. If this is the first time you set up this environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
```

```
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

1.2 About the Attacker Container

In this coursework, we should use the attacker container as the attacker machine. If you look at the Docker Compose file, you will see that the attacker container is configured differently from the other containers. Here are the differences:

- *Shared folder.* When we use the attacker container to launch attacks, we need to put the attacking code inside the container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers.

  ```
  volumes:
          - ./volumes:/volumes
  ```

- *Privileged mode.* To be able to modify kernel parameters at runtime (using `sysctl`), such as enabling IP forwarding, a container needs to be privileged. This is achieved by including the following entry in the Docker Compose file for the container.

  ```
  privileged: true
  ```

1.3 Packet Sniffing

Being able to sniff packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems. There are several different ways to do packet sniffing:

- Running `tcpdump` on containers. We have already installed `tcpdump` on each container. To sniff the packets going through a particular interface, we just need to find out the interface name, and then do the following (assume that the interface name is `eth0`):

  ```
  # tcpdump -i eth0 -n
  ```

  It should be noted that inside containers, due to the isolation created by Docker, when we run `tcpdump` inside a container, we can only sniff the packets going in and out of this container. We won't be able to sniff the packets between other containers. However, if a container uses the `host` mode in its network setup, it can sniff other containers' packets.

- Running `tcpdump` on the VM. If we run `tcpdump` on the VM, we do not have the restriction on the containers, and we can sniff all the packets going among containers. The interface name for a network is different on the VM than on the container. On containers, each interface name usually starts with `eth`; on the VM, the interface name for the network created by Docker starts with `br-`, followed by the ID of the network. You can always use the `ip address` command to get the interface name on the VM and containers.

- We can also run Wireshark on the VM to sniff packets. Similar to `tcpdump`, we need to select what interface we want Wireshark to sniff on.