Labtainer Lab Designer User Guide

June 19, 2019

Contents

1 Introduction

This manual is intended for use by lab designers wanting to create or adapt cybersecurity labs to use the Docker container-based lab framework known as "Labtainers". The Labtainer framework is designed for use with computer and network security laboratory exercises targeting Linux environments, and it is built around standard Linux Docker containers. A Labtainer exerciese may include multiple networked components, all running locally on a student's computer, but without the performance degredation associated with running multiple virtual machines.

While most Labtainer exercises focus on exploring concepts via the Linux command line – GUI based applications, e.g., browsers and Wireshark are also supported.

1.1 Benefits of Labtainers

Deploying cybersecurity labs using this framework provides three primary benefits:

1. The lab execution environment is controlled and consistent across all student computers regardless of the Linux distribution and configuration present on individual student computers. This allows each lab designer to control which software packages are present, the versions of libraries and specific configuration settings, e.g., /etc file values. These configurations may vary between labs, and they may vary between multiple computers in a single lab.

2. Assessment of student lab activity can be automated through a set of configuration files that identify expected results, thus relieving lab instructors from having to individually review detailed lab results.

3. Labs may be automatically "parameterized" for each student such that students cannot easily copy results from another student or from internet repositories.

Labtainers provide the advantages of a consistent execution environment without requiring an individual Virtual Machine (VM) per lab, and without requiring all labs to be adapted for a common Linux execution environment. These benefits can be realized whether or not labs are configured for automatic assessment, or are parameterized for each student.

Exercises that include multiple networked computers illustrate an advantage of using containers over VMs, namely, containers require significantly less resources than do VMs. A student laptop that struggles to run two or more VMs can readily run multiple containers simultaneously, as shown in this 50 second demonstration: `https://youtu.be/JDV6jGF3Szw`

Lab designers enhance labs to include automated assessment using directives built into the famework. For example, ten rather simple directives can evaluate the following question regarding a student' work on a lab depicted in Figure 1:

"Was there any single iptables configuration during which the student used nmap to demonstrate that:

- The remote workstation could reach the HTTPS port but not the SQL port, and,

- The local workstation could reach the HTTPS port and the SQL port."

1.2 Obtaining the Labtainer development kit

Installation of Labtainers is described in the *Labtainer Student Guide*, which also includes instructions for installing an Ubuntu VM (if you do not already have a Linux system), and the Labtainer framework. Our website also distributes pre-packaged VM appliances that already

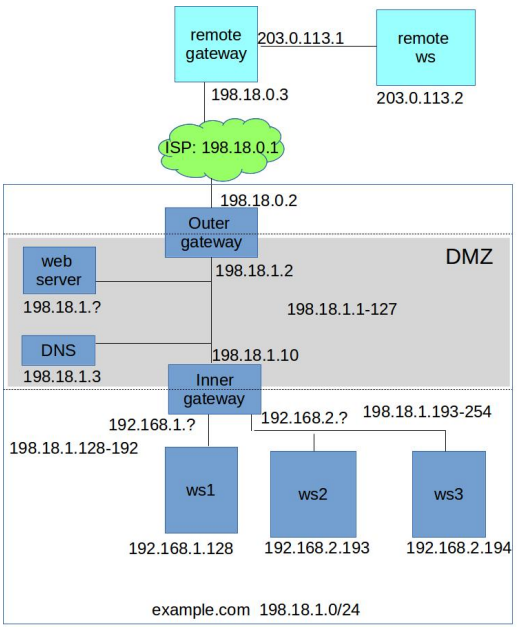

Figure 1: Example Labtainers network topology

have Labtainers installed. Labtainers will work with any Linux distribution that supports Docker containers. If you already have Docker installed on a Linux system, reference the Student Guide for other dependencies.

The difference between the development kit and the standard Labtainer distribution is primarily just the lab definition files, which are withheld from the general distribution for efficiency.

If you have a Labtainer installation (e.g., our pre-packaged VM), you can get the developer files by going to your labtainers directory, e.g., `~/labtainers/` and running `./update-designer.sh` [1] You may then want to logout and login again, or run a new `bash` shell because that script sets some environment variables.

It is suggested that you periodically run that update script to get the latest lab definition files, and to update framework software.

1.3 Content of this guide

This guide describes how to build new labs, but first, section 2 gives an overview of how students interact with Labtainers. The steps taken to create a new lab are provided in section 3, and the mechanics of defining the lab execution environment are in section 4.

Individualizing labs to discourage sharing of solutions is described in 5. Section 6 then describes how to define criteria to enable automated assessment of student work.

Networking considerations are described in 7. Section 8 covers the process of building, publishing and maintaining labs.

Strategies for creating mulit-user Labtainer exercises are discussed in section 9. Section 10 identifies limitations of the framework and section 11 includes application-specific notes, e.g., notes relavent to including Firefox in a lab.

Automated testing of labs is supported using our SimLab tool as described in Appendix A.

[1] The student password for the pre-packaged VM is "password123".

2 Overview of the student environment and workflow

Labtainers support laboratory exercises designed for Linux environments, ranging from interaction with individual programs to labs that include what appear to be multiple components and networks. Students see and interact with Linux computers, primarily via bash shell commands and GUI-based applications. In general, the Labtainer framework implementation is not visible to the student, and the Linux environment as seen by the student is not noticeably augmented to support the framework.

Labtainers are intended for use on individual student computers, e.g., a laptop, or potentially a VM allocated to the student from within a VM farm. [2] The computer utilized by a student must include the Linux operating system, e.g., as a single VM. This Linux operating system, referred to herein as the *Linux host*, can be any distribution and version which supports Docker. Students download and expand a tarball, and run an installation script as described in the *Labtainer Student Guide* [3] Alternately, students can use a Linux VM that is pre-configured with Labtainers and Docker, and is available at our website.

It is suggested that the student's Linux host be a virtual machine that is not used for purposes requiring trust. Software programs contained in cybersecurity lab exercises are not, in general, trusted. And while Docker containers provide namespace isolation between the containers and the Linux host, the containers run as privileged.

Labtainer exercises can include networking to external hosts, e.g., a Windows VM running alongside the Linux host VM, as described in section 7.2.

Students initiate any and all labs from a single workspace directory on the Linux host. To perform a specific Labtainer exercise, the student runs a *start.py* command from the Labtainer workspace, naming the lab exercise. This results in one or more containers starting up along with corresponding virtual terminals via which the student will interact with the containers. These virtual terminals typically present a bash shell. Each container appears to the student as a separate computer, and these computers may appear to be connected via one or more networks.

When a student starts a given exercise for the first time, the framework fetches Docker images from the Docker registry. Docker manages container images as a set of layers, providing efficient storage and retrieval of images having common components. The initial Labtainer installation step pulls a few baseline images (about 1.5 GB) from the public Docker registry, known as the *Docker hub*. Images for specific labs are pulled from the Docker hub by downloading only those additional layers required by that lab, and which had not been previously pulled from the hub. This is transparent to the student, other than waiting for downloads to complete.

After the student performs the lab exercise, artifacts from the container environments are automatically collected into an archive, (a zip file), that appears on the student's Linux host. The student forwards this archive file to the instructor, e.g., via email or a learning management system (LMS). The instructor collects student archive files into a common directory on his or her own Linux host, and then issues a command that results in automated assessment of student lab activity, (if the lab is designed for that), and the optional creation of an environment in which the instructor can review the work of each student.

Many cybersecurity lab exercises are assessed through use of reports in which students describe their activities and answer specific questions posed by the instructor. Labtainers are intended to augment, rather than supplant this type of reporting. The framework includes

[2] Labtainers can also support labs in which students collaborate (or compete) on shared infrastructure. Please see section 9 for information on multi-user environments.

[3] This tarball may someday be replaced by standard Linux distribution packages, e.g., Debian and/or RPM packages.

mechanisms for automating the collection of student lab reports into the artifact archive files that are collected by instructors.

3 Creating new labs

The most challenging and critical part of designing a new cybersecurity lab is the design of the lab itself, i.e., identifying learning objectives and organizing exercises to achieve those objectives. The Labtainer framework does not specifically address any of that. Rather, the framework is intended to allow you to focus more time on the design of the lab and less time on mitigating and explaining system administration and provisioning burdens you would otherwise place on students and instructors.

Typical steps for developing a new lab are:

1. Give the lab a name and create its computers using the `new_lab_setup.py` script;
2. Choose the starting baseline configuration for each computer and add software packages within a Dockerfile;
3. Define networks and connections to the lab computers in the lab's `start.config` file.
4. Populate the user's HOME directory and system directories with lab-specific files.

The remainder of this section covers the fist step and provides an example. The following section 4, covers the other three steps. After a lab is created, you can then optionally parameterize it per section 5 and/or define criteria for automated assessment per section 6

3.1 Create the first lab computer

Labtainer exercises each have their own directory under the "labs" directory in the project repository. The first step in creating a new lab within the framework is to create a directory for the lab and then cd to it. The directory name will be the name used by students when starting the lab. It must be all lower case and not contain spaces.

```
cd $LABTAINER_DIR/labs
mkdir <new lab name>
cd <new lab name>
```

After the new lab directory is created, run the "new_lab_setup.sh" script. [4]

```
new_lab_setup.py
```

This will create a set of template files that you can then customize for the new lab. These template files are referenced in the discussion below. The result of running `new_lab_setup.py` is a new labtainer lab that can be immediately run. While this new lab will initially only present you with a bash shell to an empty directory on a Linux computer, it is worth testing the lab to understand the workflow.

[4] The `$LABTAINER_DIR` will have been defined in your .bashrc file when you installed Labtainers. It should point to the `labtainers/trunk` directory. You may need to start a new `bash` shell to inherit the environment variable.

3.2 Testing the new lab

Once a new lab directory is created, and the new_lab_setup.py has been run, then you can test the new, (currently empty) lab. All student labs are launched from the labtainer-student directory. Lab development workflow is easiest if at least two terminals or tabs are used, one in the new lab directory, and one in the labtainer-student directory. So, open a new tab or window, and then:

```
cd $LABTAINER_DIR/scripts/labtainer-student
```

Then start the lab using the:

```
rebuild.py [labname]
```

command, where labname is the name of the lab you just created.

The rebuild.py command will remove and recreate the lab containers each time the script is run. And it will rebuild the container images if any of their configuration information has changed. [5] This is often necessary when building and testing new labs, to ensure the new environment does not contain artifacts from previous runs. The progress of the build, and error messages can be viewed in the labtainer.log file. While developing, it is generally a good idea to tail this log in a separate terminal:

```
tail -f labtainer.log
```

Note the "rebuild.py" command is not intended for use by students, they would use the "start.py" command. The rebuild.py utility compares file modification dates to Docker image creation dates to determine if a given image needs to be rebuilt. [6]

Stop the lab with

```
stop.py
```

When you stop the lab, a path to saved results is displayed. This is the zip file that the student will forward to the instructor.

To test adding a "hello world" program to the new labtainer, perform the following steps:

- From the new lab directory window, cd `$LABTAINER_DIR/labs/[labname]/[labname]`
- Create a "hello world" program, e.g., in python or compiled C.
- From the labtainer-student window, run rebuild.py [labname]

You should see the new program in the container's home directory. If you run the program from the container, and then stop the lab with stop.py, you will see the stdin and stdout results of the program within the saved zip file.

The "hello world" program was placed in `$LABTAINER_DIR/labs/[labname]/[labname]`. The seemingly redundant "labname" directories are a naming convention in which the second directory names one of potentially many containers. In this simple example, the lab has but one container, whose name defaults to the lab name.

The following sections describe how to further alter the lab execution environment seen by the student.

[5] The build process may generate warnings in red text, some of which are expected. These include an unreferenced "user" variable and the lack of apt-utils if apt-get is used to install packages in Dockerfiles.

[6] rebuild.py will miss file deletion. Thus, if files are deleted, you must force the rebuild using the `-f` option at the end of the rebuild.py command. Also, addition of symbolic links will not trigger a rebuild.

3.3 Multiple containers

The `new_lab_setup.sh` script can be used to create additional containers for use in the lab. For example, from your new lab directory:

```
new_lab_setup.sh -a joe_computer
```

will create a second container for your lab, named "joe_computer". If you again run the rebuild.py script, you will see two virtual terminals, each connected to one of your two independent computers. Use

```
new_lab_setup.sh -h
```

to view the operations available in that script.

The following sections describe how to configure the execution environments on your components, and how to define virtual networks connected to the components.

4 Defining the lab execution environment

A given lab typically requires some set of software packages, and some system configuration, e.g., network settings, and perhaps some lab-specific files. It can include multiple containers, each appearing as distinct computers connected via networks. The execution environment seen by a student when interacting with one of these "computers" is therefore defined by the configuration of the associated container.

Software packages are defined in each container's Dockerfile, described in the subsection below. That is followed by subsection 4.2 describing network definitions, (and other computer attributes) in the start.config file. The remaining subsections then described populating the user HOME directory and system directories, and methods for starting system services and miscellanious envrionment settings.

Labtainer containers, by default, present students with a virtual terminal and a bash shell requiring no login. Alternate initial environments, including use of the login program, are described in section 4.7.

4.1 Docker files

A default Labtainer-specific Dockerfile is placed in the new lab's "Dockerfiles" directory when the new lab is created. And additional Dockerfiles are added when the `new_lab_setup.sh -a` script adds computers to the lab. We use standard Docker file syntax, which is described at `https://docs.docker.com/engine/reference/builder/`

Simple labs should be able to use the default Dockerfile copied by the new_lab_setup.py script. That Dockerfile refers to a base Labtainer image that contains the minimum set of Linux packages necessary to host a lab within the framework. The default execution environment builds off of a recent Ubuntu image.

Each container has its own Dockerfile within the

```
$LABTAINER_DIR/labs/[labname]/dockerfiles
```

directory. The naming convention for Dockerfiles is

```
Dockerfile.[labname].[container_name].student
```

The first line of each Dockerfile identifies the baseline Labtainer image to be pulled from the Docker Hub. The initial default image is a basic Ubuntu system with a minimal set of packages. To use an alternate image having additional networking packges (e.g., tcpdump, xinetd, sshd), change the first line to:

```
FROM mfthomps/labtainer.network
```

Other alternate images include:

- labtainer.centos – A CentOS server with systemd and the true "init" initial process.
- labtainer.lamp – A CentOS server with Apache, Mysql and PHP, (the LAMP stack)
- labtainer.firefox – An Ubuntu container with the Firefox browser.
- labtainer.wireshark – The labtainer.network with wireshark added.
- labtainer.java – An Ubuntu container with the Firefox browser and the open JDK.
- labtainer.kali – A Kali Linux system with the Metasploit framework.
- labtainer.metasploitable – The Metasploitable-2 vulnerable server.

Refer to the Dockerfiles in `$LABTAINER_DIR/scripts/designer/base_dockerfiles` to see which software packages are included within each baseline image.

The Dockerfile is used to add packages to your container, e.g.,

```
RUN apt-get update && apt-get install -y some_package
```

You will also see "ADD" commands in the Docker file that populate the container directories with lab-specific files such as described in section 4.3.

Next, you must also describe your containers within the *start.config* file as described below.

4.2 Container definitions in start.config

Most single container labs can use the automatically generated start.config file without modification. Adding networks to containers and defining users other than the default "ubuntu" user requires modification of the start.config file. The following describes the major sections of that configuration file. Most of the configuration entries can be left alone for most labs.

- GLOBAL_SETTINGS – These lab-wide parameters include:
  - GRADE_CONTAINER – Deprecated
  - HOST_HOME_XFER [dir name] – Identifies the host directory via which to transfer student artifacts, relative to the home directory. For students, this is where the zip files of their results end up. For instructors, this is where zip files should be gathered for assessment.
  - LAB_MASTER_SEED [seed] – The master seed string for this lab. It is combined with the student email address to create an instance seed that controls parameterization of individual student labs.
  - REGISTRY [registry] – The id of the Docker Hub registry that is to contain the lab images. See 8 for details on the use of this keyword.
  - COLLECT_DOCS [yes/no] – Optional directive to collect lab/docs content as part of student artifacts. These are then available to the instructor in the labtainer_xfer/[lab]/docs directory. Also see 4.6.
  - CHECKWORK [yes/no] – Optional directive to disable (set to "no") ability of student to check their own work from the labtainer-student directory.

- NETWORK [network name] – One of these sections is require for each network within the lab. In addition to providing a name for the network, the following values are defined:
  - MASK [network address mask] – The network mask, e.g., 172.25.0.0./24
  - GATEWAY [gateway address] – The IP address of the network gateway used by Docker to communicate with the host. Please note that to define a different network gateway for the component, you should use the `set_default_gw.sh`. This GATEWAY field should not name the IP of any of your other components. See the the `routing_basics2` lab for examples.
  - MACVLAN_EXT [N] – Optional, causes the Docker network driver to create and use a macvlan tied to the given Nth ethernet interface (in alphabetical order) that lacks an assigned IP address. The network device is expected to be on a "host-only" VM network. The VMM should disable the DHCP server on this network. The network adaptor itself needs to be placed in primiscous mode on the Linux VM, e.g., using "sudo ifconfig enp0s8 promisc." These types of interfaces can be used to communicate with external hosts, e.g., other VMs as described in 7.2
  - MACVLAN – Similar to MACVALN_EXT, except a macvlan will not be created unless the Labtainer lab is started as a multi-user lab as descrbed in 9.
  - IP_RANGE [range] – Optional, allocates an ip range to the network, e.g., 192.168.1.4/30
- CONTAINER [container name] – One of these sections is required for each container in the lab. Default values for container sections are automatically created by the `new_lab_setup.py` script. In addition to naming the container, the following values are defined:
  - TERMINALS [quantity] – The number of virtual terminals to open and attach to this container when a lab starts. If missing, it defaults to 1. Terminal titles are set to the bash shell prompt. A value of 0 suppresses creation of a terminal, and a value of -1 prevents the student from attaching a terminal to the container.
  - TERMINAL_GROUP [name] – All virtual terminals within the same group are organized as tabs within a single virtual terminal. Terminal group names can be arbitrary strings.
  - XTERM [title] [script] – The named script is executed in a virtual terminal with the given title. The system will change to the user's home directory prior to executing the script. The script should be placed in container _bin directory, i.e.,

    ```
    $LABTAINER_DIR/labs/[labname]/[container]/_bin
    ```

    If the title is "INSTRUCTIONS", no script is necessary and the instructions.txt file in the container home directory will be displayed.
  - USER [user name] – The user name whose account will be accessed via the virtual terminals. This defaults to "ubuntu."
  - PASSWORD [password] – The password for the user name whose account will be accessed via the virtual terminals. This defaults to the user name defined above.
  - `[network name]` [ip address] – Network address assignments for each network (defined via a NETWORK section), that is to be connected to this container. A separate line should be entered for each network. The given ip address can be one of the following:
    - An IP address

* An IP address with an optional MAC address assignment as a suffix following a colon, e.g., 172.25.0.1:2:34:ac:19:0:2.
* An IP address with an optional clone offset, e.g., `172.25.0.1+CLONE` to cause each clone to be assigned an address from a sequence starting with the given address. Only intended for use with containers having the `CLONE` option described below.
* Similar to the use of the `+CLONE` suffix, `CLONE_MAC` only takes effect if the lab is started in multi-user mode. When started with the `--workstation` switch, this directs the system to generate a MAC address whose last four bytes match those of the host network interface. When stated as a multi-user lab with all containers on one VM, e.g., the `--client_count` switch, then the allocated IP address is incremeted by one less than the clone instance number.
* If `AUTO` is provided as the address, an address is chosen for you from the subnet range.

Multiple IP addresses per network interface by appending a `:n` to the `network name`, e.g.,

```
MY_LAN:1 172.24.0.3
MY_LAN:2 172.24.0.4
```

- SCRIPT [script] – Optional script to provide to the Docker create command, defaults to "bash". This must be set to "NONE" for CentOS-based components, Ubuntu systemd components, or other images that run a true Linux init process.).
- ADD-HOST [host:ip — network] – Optional addition to the /etc/hosts file, a container may have multiple ADD-HOST entries. If a network name is provided, then every component on that network will get an entry in the hosts file.
- X11 [YES/NO] – Optional, defaults to NO. If YES, the container mounts the TCP socket used by the hosts X11 server, enabling the container to run applications with GUIs, e.g., browsers or wireshark. See sql-inject as an example. See the Notes section (11) at the end of this manual for tips on using Firefox and Wireshark.
- CLONE [quantity] – optional quantity of copies of this container to create. Each copy is assigned a monotonically increasing integer starting with one, and this value can be used for the network address as describe above, and within parameterization as described in section 5. This option is not intended for use in creating multi-user labs.

A simple example of a two-container lab with network settings in the start.config file can be found in

```
$LABTAINER_DIR/labs/telnetlab
```

Entries in the start.config file can be parameterized as described in section 5, e.g., to allocate random IP addresses to components.

4.3 Lab-specific files in the student's home directory

Files that are to reside relative to the student's $HOME directory are placed in the new lab container directory. For example, if a lab is to include a source code file, that should be placed in the lab container directory. The file will appear in the student's home directory within the container when the container starts. The lab container directory is at:

```
$LABTAINER_DIR/labs/[labname]/[container name]
```

The container name in labs with a single container matches the labname by default.

All files and directories in the lab container directory will be copied to the student's HOME directory except for the _bin and _system directories. Each initial Dockerfile from the templates include this line:

```
ADD $labdir/$lab.tar.gz $HOME
```

to accomplish the copying. Except as noted below, Dockerfiles should not include any other ADD commands to copy files to the HOME directory.

4.3.1 Large or numerous files in the home directory

If there are large sized, or a high quantity of files that are to be placed relative to a container home directory, those should be placed into a "home_tar" directory at:

```
$LABTAINER_DIR/labs/[labname]/[container_name]/home_tar/
```

Use of this technique prevents these files from being collected as student artifacts, which otherwise include copies of everything relative to the home directory. This can save considerable time and space, e.g., on the instructor's computer that must collect all student artifacts. The individual files should exist in the home_tar directory, and the framework automatically creates the tar file for transfer to the Docker image, (and will do so if an existing tar file is older than any file in the directory). Manifests can be used for the home_tar content as described in 8.3. You can force collection of selected files from the home_tar by putting the filename into a file at:

```
$LABTAINER_DIR/labs/[labname]/[container_name]/_bin/noskip
```

Files whose basenames match any found in `noskip` will be collected.

Alternately, a file at /var/tmp/home.tar will be expanded into the user home directory. Use the Docker `COPY` directive to place a file here. See the

```
$LABTAINER_DIR/scripts/designer/base_dockerfiles/Dockerfile.labtainer.firefox
```

for an example. These files will not be collected unless they are newer than the original file, or if the base file name appears in the `noskip` list described above.

4.4 Lab-specific system files

All files in the

```
$LABTAINER_DIR/labs/[labname]/[container name]/_system
```

directory will be copied to their corresponding paths relative to the root directory. For example, configuration files for /etc should appear in _system/etc/.

The initial Dockerfile from the templates include this line:

```
ADD $labdir/sys_$lab.tar.gz /
```

to accomplish the copying. If a lab contains a large quantity of system files, or large files, those can be placed into the directory named:

```
$LABTAINER_DIR/labs/[labname]/[container name]/sys_tar
```

either as individual files, or in a "sys.tar" archive. In the former case, the framework will automatically create the sys.tar file. This technique can save time in building lab images becauase the files do not need to be archived for each build.

In general, files modified and maintained by the designer should go into the _system directory while static system files should go into the sys_tar directory.

4.5 System services

The Dockerfile "ENTRYPOINT" command can be used to start a system service. The general Docker model is that a single Docker container runs a single service, with logging being forwarded to the host. Labtainers disregards this model because our goal is to make a container look more like a Linux system rather than a conformant Docker container. Labtainer Dockerfiles for Ubuntu and Centos containers use systemd based images that run the /usr/sbin/init process. [7] The labtainer.network configuration of the baseline Dockerfile also starts xinetd, which will then fork services, e.g., the sshd, per the /etc/xinet.d/ configuration files.

The centos-logs lab provides an example of forcing the student to login using the traditional login program, as described in section 4.7.

4.6 Instructions for Students

Lab instructions for students can be displayed in a virtual terminal by placing an "instructions.txt" file within the home directory of one of the containers. Refer to existing labs for conventions. Additionally, textual message can be displayed to the student before any of the lab virtual terminals are created. Any text within the

```
$LABTAINER_DIR/labs/[labname]/docs/read_first.txt
```

file will be displayed on the Linux host in the terminal in which the student starts the lab. Any "LAB_MANUAL" string in that file will be replaced with the full path to a [labname].pdf file within that same docs directory. And "LAB_DOCS" is replaced by the path to the lab docs directory. One intended use is to prompt the student to open a PDF lab manual and perhaps read parts of it prior to continuing with the lab. Another intended use is to display the path to a reporting template that a student is to use for answering lab-specific questions and note taking. If the name of the symbols are prefaced by "file://", then the paths will display as links that can be opened via a right click. If the start.config file includes "COLLECT_DOCS YES", the content of the lab/docs directory will be included with the student artifacts and available extracted into the intstrutor's labtainer_xfer/[lab]/docs directory.

4.7 Running programs in Virtual Terminals

Programs can be started automatically within virtual terminals using two methods. The first is the "XTERM" directive in the container section in the start.config file described in 4.2. That is intended for programs whose results are displayed within the virtual terminal, (see the plc lab for examples). The second method is intended for user authentiation and for starting GUI based programs that will use the Linux host Xserver. If a file exists at:

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin/student_startup.sh
```

it will be executed from each virtual terminal created for the container. See the sql-inject lab and the centos-log lab examples, with the latter running the login program to require students to login prior to getting a shell prompt. [8] Note that on CentOS systems, the student_startup.sh script will be executed twice: first as root and then as the default user. Use constructs such as the following to avoid repeating operations:

[7] Now deprecated Ubuntu-based Labtainer Dockerfiles included an ENTRYPOINT command that launches a *faux_init* script that starts rsyslog, (so that system logs appear in /var/log), and runs rc.local.

[8] On CentOS systems, copy the login program from labs/centos-log/centos-log/_system/sbin/login to your container's _system/sbin directory. The login program from Ubuntu works as is.

```
id | grep root >>/dev/null
result=$?
if [[ $result -eq 0 ]]; then
   # stuff to do as root
else
   # stuff to do as default user
fi
```

it will be executed from each virtual terminal created for the container. See the sql-inject lab and the centos-log lab examples, with the latter running the login program to require students to login prior to getting a shell prompt. [9] Note that on CentOS systems, the student_startup.sh script will be executed twice: first as root and then as the default user. Use constructs such as the following to avoid repeating operations:

```
id | grep root >>/dev/null
result=$?
if [[ $result -eq 0 ]]; then
   # stuff to do as root
else
   # stuff to do as default user
fi
```

4.8 Final lab environment fixup

The initial environment encountered by the student is further refined using the optional _bin/fixlocal.sh script. The framework executes this script the first time a student starts the lab container. For example, this could be used to compile lab-specific programs afer they have been parameterized, (as described below in 5). Or this script could perform final configuration adjustments that cannot be easily performed by the Dockerfile. These scripts are per-container and reside at:

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin/fixlocal.sh
```

Note the fixlocal.sh script runs as the user defined in the start.config for the container, regardless of whether root is set as the user in the Dockerfile.
[10]

4.9 Automatic copying files from containers to the host

This feature no longer has an intended use, but it is available if you have one. Files are identified within

```
$LABTAINER_DIR/labs/[labname]/config/files_to_host.config
```

with a format of "container:filename". Any named files within the home directory of the named container will be copied to the host computer into a directory named by the lab, relative to the Labtainer working directory.

[9] On CentOS systems, copy the login program from labs/centos-log/centos-log/_system/sbin/login to your container's _system/sbin directory. The login program from Ubuntu works as is.

[10] Use of `sed -i ...` to modify configuration files (e.g., in etc), might result in overwriting symbolic links. Use `sed -i --follow-symlinks ...` to avoid that pit.

5 Parameterizing a lab

This section describes how to individualize the lab for each student to discourage sharing of lab solutions. This is achieved by defining symbols within source code or/and data files residing on lab containers. [11] The framework will replace these symbols with randomized values specific to each student. The `config/parameter.config` file identifies the files, and the symbols within those files that are to be modified. A simple example can be found in

```
$LABTAINER_DIR/labs/formatstring/formatstring/config/parameter.config
```

That configuration file causes the string `SECRET2_VALUE` within the file:

```
/home/ubuntu/vul_prog.c
```

to be replaced with a hexidecimal representation of a random value between 0x41 and 0x5a, inclusive.

This symbolic replacement occurs when the student first starts the lab container, but before the execution of the _bin/fixlocal.sh script. Thus, in the formatstring lab, the executable program resulting from the fixlocal.sh script will be specific to each student (though not necessarily unique).

5.1 Parameterization configuration file syntax

Symbolic parameter replacement operations are defined within the `config/parameter.config` file. Each line of that file must start with a `"<parameter_id> : "`, which is any unique string, and is followed by one of the following operations:

```
RAND_REPLACE : <filename> : <symbol> : <LowerBound> : <UpperBound>
  Replace a symbol within the named file with a random value within a given
  range.  The random value generator is initialized with the lab instance
  seed.

    where: <filename> - the file name (file must exist) where <symbol> is
                        to be replaced.  The file name is prefixed
                        with a container name and a ":", (the container
                        name is optional for single-container labs).
                        This may be a list of files, delimited by semicolons.
                        A file name of "start.config" will cause symbols
                        in the lab's start.config file to be replaced, e.g.,
                        to randomize IP addresses.
           <symbol> - the string to be replaced
           <LowerBound> and <UpperBound> specifies the lower and upper bound
                                        to be used by random generator
  example:

    some_parameter_id : RAND_REPLACE: client:/home/ubuntu/stack.c
        : BUFFER_SIZE : 200 : 2000
    (all one line) will randomly replace the token string "BUFFER_SIZE" found in
    file stack.c on the mylab.client.student container with a number ranging from
    200 to 2000
```

[11] An exception is the start.config file, which can be parameterized as described in the syntax description.

```
RAND_REPLACE_UNIQUE : <filename> : <symbol> : <LowerBound> : <UpperBound>
  Identical to RAND_REPLACE, except randomly selected values are never resused
  within any given upper/lower bound range.  This is intended for use on IP
  addresses, e.g., 198.18.1.WEB_IP.  It is suggested that random ranges be selected
  such that they do not intersect any non-random address allocations.

HASH_CREATE : <filename> : <string>
  Create or overwrite a file with a hash of a given string and the lab
  instance seed.
    where: <filename> - the file name that is to contain the resulting hash.
                        The file name is prefixed with a container name
                        and a ":", (the container name is optional for
                        single-container labs).
                        This may be a list of files, delimited by semicolons
                        The file name is is optionall, (in cases of a single
                        container).  This may be a
                        list of files, delimited by semicolons.
           <string> -   the input to a MD5 hash operation (after concatenation
                        with the lab instance seed)


    example:
      some_parameter_id : HASH_CREATE : client:/home/ubuntu/myseed
         : bufferoverflowinstance
    A file named /home/ubuntu/myseed will be created (if it does not exist),
    containing an MD5 hash of the lab instance seed concatentated with the
    string 'bufferoverflowinstance'.

HASH_REPLACE : <filename> : <symbol> : <string>
  Replace a symbol in a named file with a MD5 hash of a given string
  concatenated with the lab instance seed.
    where: <filename> - the file name (file must exist) where <symbol> is
                        to be replaced.  The file name is prefixed
                        with a container name and a ":", (the container
                        name is optional for single-container labs).
                        This may be a list of files, delimited by semicolons.
           <symbol> - a string that will be replaced by the hash
           <string> - a string contatenated with the lab instance seed and hashed

    example:
      some_parameter_id HASH_REPLACE : client:/root/.secret :
          ROOT_SECRET : myrootfile
    The string "ROOT_SECRET" in file /root/.secret will be replaced with an MD5 hash
    of the concatenation of the lab instance seed and "myrootfile".

CLONE_REPLACE : <filename> : <symbol> : <ignored>
  Replace a symbol with the clone instance number of a container per the CLONE option
  in the start.config file.  This is intended for use in providing instance-
  unique values on cloned containers, e.g., to assign a unique password to
  each container based on the clone number.  If the container has no clone
```

```
instance number then the symbol is replaced with an empty string.
```

The parameter_id fields may be referenced during the automated grading function, described below in section 6.3.

5.2 Synchronizing startup and parameterization

Parameterizing occurs subsequent to container "boot", but prior to running the fixlocal.sh script. The Ubuntu based images include a `waitparam.service` that delays reporting its initialization to systemd until parameterization has completed. That service is configured to run prior to rc.local. The service unit file is at:

```
trunk/scripts/designer/system/lib/systemd/system
```

If you have defined system services that should not start until parameterization has occurred, then add this to the `[Unit]` section of their service unit file:

```
After=waitparam.service
```

Note that if your `fixlocal.sh` script starts any such service, you must create a flag directory from within your `fixlocal.sh` script to unblock the waitparam.service. The following lines would achieve that:

```
PERMLOCKDIR=/var/labtainer/did_param
sudo mkdir -p "$PERMLOCKDIR"
```

5.3 Parameterizing start.config

Parameterizing of the start.config file occurs prior to Docker container creation. The framework modifies a copy of the file stored in `/tmp/start.config` and uses that when assigning attributes to containers, e.g., IP addresses. Currently only IP addresses within the start.config can parameteterized (e.g., not user names).

5.4 Simple Parameterization for Checking Own-work

The simplest, though by no means robust, strategy for ensuring students have turned in their own work, (vice getting a zip file from a friend and simply changing the name of the file), is to individualize some file on one of the containers, and then check that file and the archive file names during grading. The framework does this automatically and reports on any student archive that does not seem to have originated from a Labtainer initiated with that student's email address.

6 Automated assessment of student labs

This section describes how to configure a lab for automated assessment of student work. Note the framework does not require automated assessment, e.g., the "results" of a lab may consist entirely of a written report submitted by the student. Support for automated collection of written reports is described in 4.6 and the use of COLLECT_DOCS in the start.config file.

The goal of automated assessment is to provide instructors with some confidence that students performed the lab, and to give instructors insight into which parts of a lab students may be having difficulty with. The automated assessment functions are not intended to standardize each student's approach to a lab, rather the goal is to permit ad-hock exploration by students.

Therefore, lab designer should consider ways to identify evidence that steps of a lab were performed rather than trying to identify everything a student may have done in the course of the lab.

Automated assessment is achieved by first generating artifact files while the student works. That is described in the first subsection below. Next, artifacts within those files are identified as described in section 6.2. The values of the resulting artifacts are then compared to expected values, as per section 6.3.

6.1 Artifact files

The files from which artifacts are derived include persisent data, such as system logs and `.bash_history`, as well as timestamped snapshots of transitory data such as stdout of a program. Lab designers can also generate customized artifacts in response to student actions using scripts that automatically execute when selected programs or utilities are executed – or when selected files are accessed. The following paragraphs describe how these artifacts are generated.

The Labtainer framework use of timestamps allows designers to express temporal relationships between artifacts, and thus between events. For example, the designer can determine if two distinct artifacts were part of the same stdout stream. Or if artifacts in the stdout stream from one program were occuring during the invocation of a different program that generated other specific artifacts. The framework also can incorporate timestamps from standard log file formats, e.g., syslog, allowing the designer to determine if some logfile entry occurred during the invocation of a program whose stdout stream contains selected artifacts. As a more concrete example, the use of timestamps allows the designer to determine that a spcific web log record occurred during invocation of some program that produced a specific artifact.

6.1.1 Capturing stdin and stdout

Each time the student invokes a selected program or utility, the framework captures copies of standard input and standard output, (stdin and stdout) into timestamped file sets. This is transparent to the student. (Also see the following section for capturing program output other than stdout.) These timestamped file sets, selected system logs, and everything relative to the student's home directory, are automatically packaged when the student completes the lab. These packages of artifacts are then transferred to the instructor, (e.g., via email or a LMS), and ingested into the instructor's system where lab assessment occurs. Timestamped stdin and stdout files are captured in `$HOME/.local/result`

By default, stdin and stdout for all non-system programs is captured, e.g., the results of an "ls" command are not captured. The stdin and stdout of system programs[12] will be captured if the program names appear at the beginning of a line in the *treataslocal* file at

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin/treataslocal
```

The basename of the treataslocal entries are compared to the basename of each command. [13] Starting of services can be monitored through use of treataslocal entries having a ".service" suffix, e.g., `httpd.service` would generate timestamped artifact files whenever httpd was started (or restarted) using systemctl, service or /etc/init.d/... See section 6.4.4 for the intended use of this feature.

Non-system programs can be excluded from stdin/stdout capturing by including their names in a "ignorelocal" file in that same directory. [14]

[12] The "source" directive is not a system program, and should not be included in a treataslocal file.

[13] In other words, if the treataslocal entry is: `usr/bin/nmap`, the path leading to nmap is ignored.

[14] These should not include path information, just the program name.

6.1.2 Capturing program file output

Sometimes program file output is of interest to automated assessment, e.g., the program may not have useful stdout. The treataslocal entries can include optional output file identifiers that cause timestamped copies of specified files to be made whenever the named program terminates. If program file output from local programs is to be captured in timestamp files (in addition to the stdout and stdin), simply include those program names in the treataslocal file. These output file identifiers are of the form:

```
program    delim_type:delim_value
        where delim_type is one of:
            starts -- the output file name is derived from the
            substring following the given delim_value within the
            command line typed by the student.  For example,
            "dd starts:of=" for a command line of
            "dd in=myfs.img of=newfile" would yield an output
            file name of "newfile".

            follows -- the output file name is the command line
            token following the given delim_value.  For example,
            "myprogram follows:myprogram" for a command line of
            "myprogram outfile" would yield "outfile" as the output
             file name.

            file -- the delim_value is the output file name
```

The resulting timestamped files are located with the stdin and stdout files in .local/result

6.1.3 Bash History

The framework collects all student bash history into the `$HOME/.bash_history` and `/root/.bash_history` files. These files are available for reference as an artifact file.

6.1.4 System logs

All files referenced in the `results.config` file, (described below in section 6.2 will be collected into the artifact archive.

6.1.5 Capturing information about the environment

Some labs require the student to alter system configuration settings, e.g., using the sysctl command to affect ASLR. A *precheck.sh* script in:

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin
```

is intended to contain whatever commands are necessary to record the state of the system at the time a program was invoked. The stdout of the precheck.sh script is recorded in a timestamped *precheck.stdout* file. The timestamp of this file will match the timestamp of the stdin and stdout artifacts associated with the command that caused precheck.sh to run. The precheck.sh is passed in the full path of the program as an argument, thereby allowing the designer to capture different environment information for different commands.

As another example, consider the file-deletion lab *precheck.sh* script. It mounts a directory, lists its content, and unmounts it. This all occurs transparently to the student, and, in this

example, helps confirm a specific file was in fact deleted at the time of issuing a command to recover deleted content from the volume.

In other situations, you may wish to capture environment information when selected commands are executed, even though you have no interest in stdin or stdout of those commands. For example, imagine you want to capture the file permissions of /usr/bin/tcpdump whenever that command is executed. This can be achieved by including /usr/bin/tcpdump in a list within a file at:

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin/forcecheck
```

and then include `ls -l /usr/bin/tcpdump` in the precheck.sh script. Note that the *forcecheck* list of programs must include the full path name. The *forcecheck* file can be used instead of a *treataslocal* file entry for those cases where stdin and stdout are not required for goal assessment. An example of the use of forcecheck can be found in the *capabilities* lab.

6.1.6 Capturing file access events

File creation, reading and modification events can be recorded using a combination of a `notify` file and an optional `notify_cb.sh` script at:

```
$LABTAINER_DIR/labs/[labname]/[container name]/_bin/
```

The `notify` file will name directory or file paths and the access modes of interest, one entry per line, having this format:

```
<file_path> <mode> [output file]
```

where the `file_path` is the absolute path to the file of interest, and `mode` is one of the following:

- `CREATE` Assumes the path is to a directory. This will capture any file or directory creation within the named directory.
- `ACCESS` will capture any read of the file named by the path.
- `MODIFY` will capture any write to the file named by the path.

The optional `output file` will be used for the timestamped filename of the output from the event (instead of the default `notify.stdout.YYMMDDHHMM` Each time an event occurs matching a criteria specified in the `notify` file, the `notify_cb.sh` script is invoked (if it exists), passing in the given path and access mode. The script is also provided with the most recent command issued by either the container user, or the root account (whichever is more recent). If there is no `notify_cb.sh` script, then the output consists of the file_path, the most recent command, and the associated user (e.g., root). See the `acl` lab for an example.

The output from the notify event is captured in timestamped files, just as those resulting from events described in Section 6.1.5. If the optional output file provided in the `notify` list is given as `precheck`, then events resulting from program invocation, e.g., due to use of a `forcecheck` file, can be recorded in the very same timestamped file as events resulting from a `notify` file. In such cases, output from the former will preceed output from the latter within the file. The framework will append the `notify` output to any timestamped `precheck.stdout` file that was created up to two seconds prior to the `notify` event. Inclusion of both outputs into one timestamped file allows the designer to identify events that occured as part of a single program invocation. Again, see the `acl` lab for an example.

6.1.7 Generating results upon stopping the lab

The lab designer can cause a script to run on selected containers whenever the student stops a lab. This is achieved by creating an script or executable program at:

`trunk/labtainers/lab/<lab>/_bin/prestop`

The stdout of any such program will be written to a timestamped file named `prestop.stdout.timestamp` The framework ensures that all such scripts on all of the lab containers will complete prior to shutting down any of the containers, and all the timestamps will be the same. Note the Labtainers framework generally allows students to acheive their goals at any point in their exploration, and the labs typically do not require the student to leave the system in any particular state. In other words, students should be free to continue experimenting subsequent to getting the correct results. Thus, any use of the prestop feature should be accompanied by a lab manual entry advising the student that they may restart a lab after issuing the `stoplab` command. [15]

6.1.8 Artifact archives

Artifacts from student labs are combined into a zip file that is placed in the student transfer directory, typically at `/labtainer/xfer/<labname>`. Students provide this file to their instructor for automated assessment, e.g., via email or an LMS.

Other uses for student artifacts are facilitated through use of a script named:

`labtainers/labs/<lab>/bin/postzip`

If such a script exists, it is executed after all of the student artifacts are zipped up. See the cyberciege lab for an example of postzip processing.

6.2 Artifact result values

The automated assessment functions encourage labs to be organized into a set of distinct "goals". For each goal, the lab designer identifies one or more specific fields or attributes of artifact files that could be compared to "expected" values. These lab-specific artifacts are identified within the file at:

`labtainer/trunk/labs/<lab>/instr_config/results.config file`

Artifact files are identified in terms of:

1. The program that was invoked
2. Whether the artifact is in stdin or stdout or is program output (prgout) as descrbed in section 6.1.2
3. An explicit file name, either as an absolute path or relative to the user HOME directory. These are intended to be persistent log files, e.g., syslogs.

One or more properties of each artifact file are assigned symbolic names, referred to herein as *results*, which are then referenced in the goals.config file to assess whether results match expected values. Directives within the results.config file assign each result a value having one of three types:

- Boolean, e.g., did an artifact file contain a specific string or regular expression?

[15] Perhaps a `goalsmet` type of command should be added that does nothing but record prestop results without actually stopping the lab?

- String, e.g., the third space-delimited token on the first line containing the string "Audience says:"
- Numeric such as the quantity of lines in an artifact file, or the quantity of occurances of a string in an artifact file.

There are typically multiple instances of each result, each with its own associated timestamp. The framework automatically associates timestamps with results, thereby allowing the designer to express temporal relationships between results as introduced in section 6.1 The timestamp associated with any given result will be derived from different sources depending on the nature of the results.config directive:

- The timestamp of the artifact file. For example, each stdout artifact file name includes a timestamp reflecting when the program was invoked, (and its corresponding stdin file contains an entry reflecting when the program terminated).
- A timestamped entry from a log file, e.g., an entry in a web log, that matches criteria specified in the results.config directive.
- The invocation times of `time_delimiter` programs, syntactically associated with system log artifact files. This allows designers to temporally group syslog results that were generated between changes to system configurations as defined by invocation of the `time_delimiter` program, e.g., a script that alters the routing table. See section 6.4.4 for additional information.

Directives within the results.config file each have the following format:

```
<result> = <file_id> : <field_type> : <field_id> [: <line_type> : <line_id>]
    where:
          result -- The symbolic name of the result, which will be referenced in
                    the goals configuration file.  It must be alphanumeric,
                    underscores permitted.
          file_id -- Identifies a single file, or the set of files to be parsed.
                The format of this id is:
                   [container_name:]<prog>.[stdin | stdout | prgout] |
                         [container_name:]file_path[:time_delimiter]
                 where <prog> is a program or utility name whose stdin, stdout,
                 or program output (prgout) artifacts will include timestamps.
                 The optional container_name identifies the container hosting
                 the file.  Labs with a single container can omit this qualifier.
                 Alternately, an explicit
                 file_path is intended for log files of services that persist
                 across multiple student operations.  If the given path is not
                 absolute, it is relative to the container user's home directory.
                 The wildcard character '*' can be used in place of <prog>,
                 i.e., *.stdin is for all stdin artifacts and *.stdout is for all
                 stdout artifacts.  The optional time_delimiter qualifier is
                 explained further below.
          field_type - Optional, defaults to "TOKEN", possible values include:
             TOKEN      -- Treat the line as space-delimited tokens
             PARENS     -- The desired value is contained in parenthesis
             QUOTES     -- The desired value is contained in quotes
             SLASH      -- The desired value is contained within slashes,
```

```
                e.g., /foo/
   LINE_COUNT -- The quantity of lines in the file. Remaining fields
                are ignored.
   CHECKSUM   -- The result value is set to the md5 checksum
                of the file.
   CONTAINS   -- The result value is set to TRUE if the file
                contains the string represented in field_id.
   FILE_REGEX -- The result value is set to TRUE if the file
                contains the regular expression represented in field_id.
                The python findall function is used on the entire file.
                See the acl lab for an example of multi-line expressions.
   LOG_TS     -- Used with timestamped log files, this results in a
                timestamped set of boolean results with a value
                of TRUE for each log line that contains the string
                represented in the field_id.
   FILE_REGEX_TS Like LOG_TS, but uses regular expressions.
   LOG_RANGE  -- Similar to LOG_TS, except the timestamped entries
                are ranges delimited by the matching log entries.
   STRING_COUNT--The result value is set to the quantity of
                occurances of the string represented in field_id.
   COMMAND_COUNT-Intended for use with bash_history files, counts
                the occurances of the command given in the field_id.
                Commands are evaluatd considering use of sudo, time, etc.
   PARAM      -- The result value is set to nth parameter
                (0 is the program name), provided in the
                program invocation.

   SEARCH     -- The result is assigned the value of the search
                defined by the given field_id, which is treated as an
                expression having the syntax of pythons parse.search
                function.  E.g., "frame.number=={:d}" would
                yield the frame number.
   GROUP      -- Intended for use with "REGEX" line types, the
                result is set to the value of the regex group
                number named by the field_id.  Regular expressions
                and their groups are processed using the python
                re.search semantics.
   TIME_DELIM -- The timestamps of the named files are used to create
                a set of time ranges, e.g., for use in time_during
                goal operators.


field_id -- An integer identifying the nth occurance of the field type.
            Alternately may be "LAST" for the last occurance of the
            field type, or "ALL" for the entire line (which causes the
            field type to be ignored).  Or if field_type is SEARCH, the
            field_id is treated as the search expression.
            If field_type is "CONTAINS", the
            remainder of the line is treated as a string to be
            searched for.  If field_type is "PARAM", the field_id is
```

```
                the 1-based index of the parameter whose value is to be
                assigned, and no other fields should be present.
                If field_type is "CHECKSUM", no other field is required.
    line_type - Identifies how the line is to be identified, values include:
        LINE           --  The line_id is an integer line number
                           (starting at one).  Use of this to identify
                           lines is discouraged since minor lab changes
                           might alter the count.
        STARTSWITH     --  the line_id is a string.  This names the
                           first occurrence of a line that starts with
                           this string.
        HAVESTRING      -- The line_id is a string.  This names the
                           first occurrence of a line that contains the
                           string.
        REGEX           -- The line_id is a regular expression.  This names the
                           first occurrence of a line that matches the regular
                           expression.  Also see the "GROUP" field_type.
        NEXT_STARTSWITH -- the line_id is a string.  This names the
                           line preceeding the first occurrence of a line
                           that starts with this string.
        HAVESTRING_TS   -- Intended for use with log files that have
                           timestamped entries.  Each entry containing
                           the string identified in line_id will have
                           its result stored as a timestamped value
                           as if it came from a timestamped stdout or
                           stdin file.  See the snort lab for an example.
        REGEX_TS        -- Similar to HAVESTRING_TS, but with REGEX semantics,
                           including optional use of the GROUP field_type.
    line_id - See line_type above.  String values, e.g., for "STARTSWITH"
              can be a parameterized value from the param.config file.
              Preface these with a "$".
```

6.2.1 Converting artifact file formats

Some artifact file formats are not easily referenced by results.config directives. For example, a browser history file in the .sqlite format is binary. Such files can be processed into a more convenient form through use of a script at:

```
$LABTAINER_DIR/labs/[lab]/instr_config/pregrade.sh
```

Modify or expand on the default pregrade.sh script. In general, the pregrade.sh script is expected to extract or convert data from an artifact file, and write it into a new file in the .local/results directory of the container.

6.3 Evaluating results

Results of student lab activity are assigned symbolic names by the results.config file as described above. These results are then referenced in the goals.config to evaluate whether the student obtained expected results. Most lab goals defined in the goals.config file will evaluate to TRUE or FALSE, with TRUE reflecting that the student met the defined goal. In addition to these binary goals, the designer can capture and report on quantities of events, e.g., the number

of times a student ran a specific program. Once evaluated, a goal may determine the value of subsequent goals within the goals.config file, i.e., through use of boolean expressions and temporal comparisons between goals. The evaluated state of each goal can then contribute to an overall student assessment.

Student results may derive from multiple invocations of the same program or system utility. The framework does not discourage students from continuing to experiment and explore aspects of the exercise subsequent to obtaining the desired results. In general, Labtainer assessment determines if the student obtained expected results during any invocation of a program or system utility, or during a time period delineated by timestamp ranges described in section 6.2. [16]

The goals.config file contains directives, each of which assigns a value to a symbolic name referred to as the `goal_id`. Each goal_id may have multiple instances of timestamped values, with their associated timestamp ranges inherited from results. Examples of assigning values to a goal_id include:

- A goal_id is automatically created for each boolean result from the results.config file. The timestamps are directlly inherited from the results.
- The value of a specific result is compared (e.g., do two strings match?) to a literal expected value. A boolean goal_id value is generated for each referenced result's timestamp.
- The value of a specific result is compared to a parameterized value generated from the student email address as described in section 5. A boolean goal_id value is generated for each referenced result's timestamp.
- Timestamps and boolean values of two different goal_id's are compared. For example, "was a TRUE value for `result A` generated while a TRUE value for `result B` was being generated?" A boolean goal_id is generated for each timestamp range of `result B` within which falls at least one `result A` timestamp.
- A boolean expression consisting of multiple `goal_id`'s and boolean operators such as OR, AND, NOT_AND. A boolean goal_id is generated for each timestamp range for which there is an instance of every goal_id named in the expression.

The following syntax defines each goal within the goals.config file. While the syntax may appear complex, most goals can be expressed simply as can be seen in section 6.4 and in the Labtainer exercises distributed with the framework.

```
<goal_id> = <type> : [<operator> : <resulttag> : <answertag> | <boolean_expression>
            | goal1 : goal2 | <resulttag> | value : subgoal_list]
  Where:
    <goal_id> - An identifer for the goal. It must be alphanumeric
                (underscores permitted).
    <type> - must be one of the following:
         matchany    - Results from all timestamped sets are evaluated.
                       If the answertag names a result, then both that
                       result and the resulttag must occur in the same
                       timestamped set.  The 'matchany' goals are treated
                       as a set of values, each timestamped based on the
                       timestamp of the reference resulttag.
```

[16] In those cases where the student is required to obtain the expected results during the final invocation of a program, the *matchlast* goal type may be specified as described below.

```
matchlast   - only results from the latest timestamped set are
              evaluated.
matchacross - The resulttag and answertag name results.  The
              operator is applied against values in different
              timestamped sets.  For example, a "string_diff"
              operator would require the named results to have
              at least two distinct values in different
              timestamped sets.  Note: 'matchacross' cannot
              be used within the boolean expression defined below.
boolean     - The goal value is computed from a boolean expression
              consisting of goal_id's and boolean operators, ("and",
              "or", "and_not", "or_not", and "not"), and parenthisis
              for precedence.  The goal_id's must be from goals defined
              earlier in the goals.config file, or boolean results
              from results.config.  The goal evalutes to
              TRUE if the boolen expression evaluates to TRUE for any
              of the timestamped sets of goal_ids, (see the 'matchany'
              discussion above).  The goal_id's cannot include any
              "matchacross" goals.  NOTE: evaluation is within
              timestamped sets.  If you want to evaluate across
              timestamps, use the count_greater_operator below.
count_greater The goal is TRUE if the count of TRUE subgoals in the
              list exceeds the given value.  The subgoals are
              summed across all timestamps.  The subgoal list is
              comma-separated within parenthesis.
time_before - Both goal1 and goal2 must be goal_ids from previous
              matchany, or boolean values from results.config
              A timestamped goal is created for each goal2
              timestamped instance whose timestamp is proceeded
              by a goal1 timestamped instance. The goal for that
              timestamp will be TRUE if the goal2
              instance is TRUE, and at least one of the goal1
              instances is TRUE. These timestamped goals can
              then be evaluated within boolean goals.
time_during - Both goal1 and goal2 must be goal_ids from previous
              matchany goal types, or boolean values from
              results.config.  Timestamps include a start and end
              time, reflecting when the program starts and when it
              terminates.  A timestamped goal is created for each
              goal2 range that encompasses a goal1 timestamp.
              The goal for that timestamp will be TRUE if the
              goal2 instance is TRUE, and at least one goal1 instance
              is TRUE. These timestamped goals can then be
              evaluated within boolean goals.
time_not_during Similar to time_during, but timestamped goals are
              always created for each goal2.  Each such goal is True
              unless one or more goal1 times occur within a True goal2
              range.
execute     - The <operator> is treated as a file name of a script to
```

```
                        execute, with the resulttag and answertag passed to the
                        script as arguments.  The resulttag is expected to be
                        one of the symbolic names defined in the results.config
                        file, while the answertag is expected to be a literal
                        value or the symbolic name in the parameters.config file
                        Note: the answertag cannot be a symbolic name from
                        results.config
        count       - If the remainder of the line only includes a resulttag,
                        then the goal value is assigned the quanity of
                        timestamped files containing the given resulttag.
                        Otherwise the goal value is assigned the
                        quantity of timestamped files having results
                        that satisfy the given operator and arguments.
        value       - The goal value is assigned the given resulttag value from
                        the most recent timestamped file that contains the resulttag.

<operator> - the following operators evaluate to TRUE as described below:
   string_equal -  The strings derived from <answertag> and <resulttag>
                     are equal.
   string_diff -   The strings derived from <answertag> and <resulttag>
                     are not equal.
   string_start -  The string derived from <answertag> is at the start of
                     the string derived from <resulttag>.
           example:  answertag value = 'MySecret'
                     resulttag value = 'MySecretSauceIsSriracha'
   string_end -    The string derived from <answertag> is at the end of
                     the string derived from <resulttag>.
           example:  answertag value = 'Sriracha'
                     resulttag value = 'EatMoreFoodWithSriracha'
   string_contains The string derived from <answertag> is contained within the
                     string derived from <resulttab>.
   integer_equal - Integers derived from <answertag> and <resulttag>
                     are equal.
   integer_greater - The integer derived from <answertag> is greater than
                       that derived from <resulttag>.
   integer_lessthan- The integer derived from <answertag> is less than
                       that derived from <resulttag>
   <executable_file> - If the type is 'execute' then <operator> is a filename of
                       an executable.

<resulttag>  -- One of the symbolic names defined in the results.config file.
                The value is interpreted as either a string or an integer,
                depending on the operator as defined above.  Alternately,
                for integer operators within matchany types,  this
                may be an arithmetic expression within parentheses.  For example,
                "(frame_number-44)".

<answertag>  -- Either a literal value (string, integer or hexidecimal), or a
                symolic name defined in the results.confg file or the
                parameters.config file:
```

```
answer=<literal>    -- literal string, integer or hex value
                       (leading with 0x), interpretation depending
                       on the operator as described above.
result.<symbol>     -- symbol from the results.config file
parameter.<symbol>  -- symbol from the parameters.config file
parameter_ascii.<symbol> -- same as above, but the value parsed as
                       an integer or hexidecimal and converted to an
                       ascii character.
```

Note that values derived from the parameters.config file are assigned the same values as were assigned when the lab was parameterized for the student.

6.3.1 Assessment Report

Evaluation of student results occurs on an instructor container, via a script named `instructor.py`, which runs automatically when the instructor runs the `gradelab [lab]` command. The script can also be run manually, e.g., to test changes and additions to grading configuaration files. It must be run from the HOME directory on the container that results from running the gradelab command with the `-d` option [17]. By convention, all goals and boolean results whose symbolic names are not prefaced with an underscore (_), will have corresponding entries in the assessment report, located in the home directory in a file named `<lab name>.grades.txt`

6.3.2 Document the meaning of goals

Instructors will see descriptions of lab goals when they start the lab using `start.py`. These descriptions are embedded within comments in the goals.config and results.config files. The descriptions are associated with symbolic names that immediately follow the documentation directives as described below:

```
# SUM:  -- The remainder of the line and comment lines that immediately
           follow are displayed independent of any goal symbols.
# DOC:  -- The remainder of the line and comment lines that immediately
           follow are displayed for the symbolic name that follows
           the comment lines.
# GROUP:-- The remainder of the line and comment lines that immediately
           follow are displayed for the group symbolic names that
           follows the comment lines.  The group ends on a blank line, or
           new comment.
```

See existing labs for examples.

6.4 Assessment examples

The following examples illustrate some typical assessment operations as they would be defined in the results.config and goals.config files.

[17] Be sure to run stopgrade after use of the `-d` option to shut down the grading container when you are done with it

6.4.1 Do artifact files contain specific strings?

Consider the labs/formatstring/instr_config/results.config file for a few examples. The first non-comment line defines a result having the symbolic name "_crash_sig":

```
_crash_sig = vul_prog.stdout : CONTAINS : program exit, segmentation
_crash_smash = vul_prog.stdout : CONTAINS : *** stack smashing detected
```

This result is TRUE for each timestamped stdout file resulting from running the vul_prog program in which the file contains the string "program exit, segmentation". The goals.config includes this goal:

```
crash =  boolean : ( _crash_smash or  _crash_sig )
```

The value of the crash goal is TRUE if either result was ever true. Use of the `count_greater` operator in the above example would also provide the desired assessment. Note that the boolean operator only assesses values within timestamped sets. For example, if the result values came from different program outputs, then they may not be within the same timestamp, and thus would not compare. In such a case, the `count_greater` operator should be used.

6.4.2 Compare value of a field from a selected line in an artifact file

Again refrence the labs/formatstring/instr_config/results.config file. The third non-comment line defines a result having the symbolic name "origsecret1value":

```
origsecret1value = vul_prog.stdout : 6 : STARTSWITH : The original secrets:
newsecret1value = vul_prog.stdout : 6 : STARTSWITH : The new secrets:
```

The timestamped results are found by looking at stdout from the "vul_prog" program, and finding the first line that starts with: "The original secrets:". The result is assigned the value of the sixth space-delimited token in that line. The "newsecret1value" assignment is similar. The goals.config file includes:

```
modify_value = matchany : string_diff : newsecret1value : result.origsecret1value
```

, which will be TRUE if any of the vul_prog stdout files include a "newsecret1value" that differs from its "oldsecret1value".

6.4.3 My desired artifacts are not in stdin or stdout, the program outputs a file

See section 6.1.2

6.4.4 Distinguish log file entries generated before and after configuration changes

When log files are named in results.config files, you can qualify the log file name with the name of program whose invocation serves as a `time_delimiter` introduced in section 6.2. The `time_delimiter` identifies some monitored program whose start times will be used to organize the log file into a set of timestamped results. This differs from use of HAVESTRING_TS and REGEX_TS in that those store results as descrete timestamped values for each timestamp found in the log file. The `time_delimiter` timestamp values are based on the start times of the monitored program. The intended use is to group results from programs whose actions result in entries in the log file. This can be useful for grouping system log entries based on system configuration changes (i.e., accomplished via the `time_delimiter` program).

Consider a lab that directs students to alter iptables on a component using the /etc/rc.local script. The student is required to demonstrate a desired iptables configuration by running

nmap on various other components. The instructor wants to confirm that some set of expected stdout from nmap running on different components all occurred within a single configuration of iptables, delimited by the running of rc.local. In other words, the student cannot succeed by altering iptables between invocations of nmap on different components. In this example, the `file_path` would name the iptables log, and the `time_delimiter` would name rc.local. In order to track invocations of rc.local, we would add it to the `treataslocal` file described in section 6.1.1.

Then, if the lab results.config were:

```
_iplog = outer_gw:/var/log/ulog/syslogemu.log:rc.local : \
         CONTAINS : IPTABLES DROPPED
_remote_nmap_443 = remote_ws:nmap.stdout : CONTAINS : 443/tcp open  https
_remote_nmap_sql = remote_ws:nmap.stdout : CONTAINS : 3306/tcp open  mysql
_local_nmap_443 = ws1:nmap.stdout : CONTAINS : 443/tcp open  https
_local_nmap_sql = ws1:nmap.stdout : CONTAINS : 3306/tcp open  mysql
```

The `_iplog` result would then have up to N+1 timestamped instances, where N is the quantity of times that rc.local was executed. The first possible timestamp would have a starting time of zero and an ending time of the very first invocation of rc.local. The nmap results would each have timestamps corresponding to their times of execution. Note the nmap results include results from two different computers, ws1 and remote_ws.

A goals.config file of:

```
remote_nmap_443 = time_during : _remote_nmap_443 : _iplog
remote_nmap_sql = time_during : _remote_nmap_sql : _iplog
local_nmap_443 = time_during : _local_nmap_443 : _iplog
local_nmap_sql = time_during : _local_nmap_sql : _iplog
remote_correct = boolean : ((remote_nmap_443 and_not remote_nmap_sql) \
                            and local_nmap_443 and local_nmap_sql)
```

would generate sets of nmap goals grouped into timestamps corresponding to the `_iplog` results. The `remote_correct` boolean expression could then be read as: "Was there any single iptables configuration during which the student used nmap to demonstrate that:

- The remote workstation could reach the HTTPS port but not the SQL port, and,
- The local workstation could reach the HTTPS port and the SQL port.

6.4.5 Delimiting logs by starting services

Another example of the use of `time_delimiter` log file qualifiers is a web server, and its corresponding httpd log. It may be desired to group log entries generated during a single configuration of the web server, delimited by the starting of the web server, e.g., via `sudo systemctl restart httpd`. Here, our `time_delimiter` program is the use of systemctl to start or restart the httpd. Services are named in the `treataslocal` file by giving them a suffix of ".service", e.g.,

```
httpd.service
```

and that same name is used for our `time_delimiter`, e.g.,

```
web_log = vuln-site:/var/www/csrflabelgg.com:httpd.service : CONTAINS : GET / HTTP/
```

will create time stamp ranges delimited by the starting of the web server. Results from other programs, (or other results derived from the web log), could then be similarly group using the `time_during` operation, e.g.,

```
_some_goal = time_during : other_result : web_log
_some_other_goal = time_during : yet_another_result : web_log
success = boolean (_some_goal and _some_other_goal)
```

The success goal would only then be TRUE if the two goals each occurred during a single instance of the web server configuration, as delimited by use of `systemctl restart httpd`.

6.4.6 Delimiting time using log file entries

An alternate way to create groupings of log file entries is to use the log_range result type. For example, a result.config directive of:

```
syslog_slices = /var/log/messages : LOG_RANGE : Started System Logging Service
```

would create a set of timestamped values whose ranges are based on occurrences of the system logging service being started. The use of `time_during` and/or `time_not_during` and `boolean` in the goals.config could then be used to assess whether two or more events occurred during a given system log configuration. See the centos-log lab for an example.

6.5 Debugging automated assessment in labs

Developing automated assessment for a new lab typically requires some amount of debugging. This section is intended to guide new developers through the process.

When the `gradelab` script is run from labtainers-instructor, the configuration files in labs/[lab name]/instr_config are validated. If syntax errors are found, error messages are displayed at the terminal and processing halts. The error messages identify the offending result.config or goals.config entry. Refer to sections 6.2 and 6.3 for the expected syntax of these files.

Once initial syntax checking is passed, the lab is graded for each student. If the grading table does not display, or it displays incorrects values, then find run the gradelab command with the `-d` option. At the resulting terminal, enter the `instructor.py` command. That may display diagnostics at the terminal. It will also generate a /tmp/instructor.log file of debugging messages.

At this point, the workflow is easiest if you edit/test from that container – just remember to transfer your revised .config files from the container before doing a `gradelab [lab] -r`! A copy of your config files are in `~/.local/instr\_config`. You can edit those and try running instructor.py again. The [lab].grades.json contains the results of the goals assessment. You can find the results assessment for each student beneath the directory whose name is prefaced with the student email. From there, look in .local/result to find json files reflecting intermediate results of assessing the student results. The actual student result arifacts can be found in `~/[student dir]/[lab].[container].student/.local/result`.

The mechanics of performing the lab (so that you can test grading for different outcomes) can be automated using the SimLab tool described in Appendix A.

7 Networking

Most networking is simply a matter of defining networks and assigning them to containers as described in 4.2.

In addition to networks properties defined in the start.config file, each container `/etc/host` file includes a "my_host entry" that names the host Linux. By Docker default, each container includes a default gateway that leads to the Linux host. This allows students to scp files to/from the container and host. It also allows the student to reach external networks, e.g., to fetch additional packages in support of student exploration.

In many instances, the lab designer will want to define a different default route for a container. Each container includes a *set_default_gw.sh* script that can be added to the */etc/rc.local* (or *fixlocal.sh*) file to redefine the default gateway. This script will automatically retain a route table entry so that the student can reach the "my_host" address. Additionally, those baseline images include a *togglegw.sh* script that the student can use to toggle the default gateway between one that leads to the host, and one defined for the lab. This allows students to add packages on components having lab-specific default gateways.

7.1 Realistic Network Routing and DNS

Some labs will strive to represent realistic networking environments, e.g., several networked components including gateways and DNS servers. To achieve that, you must override Docker, which automatically sets the container's /etc/resolv.conf file to use the host system DNS resolution. This is in addition to the default routes described above. While convenient, these mechanisms can distract and confuse students, particularly when routing and DNS resolution are central to the point of the exercise, (e.g., a DNS cache poisoning lab).

These Docker defaults can be easily overridden to present a more realistic networking environment. A worked example of such a topology can be seen in the *routing-basics* lab. This lab includes the following properties that can be reproduced in other labs:

- Default routes to gateway components.
- DNS definitions in /etc/resolv.conf that name gateway components.
- Use of `iptables` in gateway components to implement NAT.
- A hidden ISP component that exchanges network traffic with the host Linux system, thereby allowing all visible components to include routing, DNS and iptables entries that do not expose virtual networking tricks. See section 11.6 for additional information.

7.2 Communicating with external hosts or VMs

A container can be configured to support network communication hosts external to Labtainers. For example, consider a VirtualBox VMM that hosts a Linux VM that runs Labtainers and a Windows VM. Assume the Windows VM has a fixed IP of 192.168.1.12 and the container will be assigned a network address of 192.168.1.2. To permit network communication between a container on the Linux VM and the Windows VM:

- Define a host-only network in VirtualBox with ip address/mask 192.168.1.1/24, and assign that to the two VMs, configuring the VM network links to use promiscuous mode.
- Start both VMs. On the Linux VM, make note of the ethernet interface associated with the host-only network (assumed to be enp0s8 below).
- On the Linux VM, ensure the network interface is in promiscuous mode, `sudo ifconfig enp0s8 promisc`
- In the container start.config file, define a network as:

```
NETWORK  LAN
     MASK 192.168.1.0/24
     GATEWAY 192.168.1.1
     MACVLAN_EXT 1
     IP_RANGE 192.168.1.0/24
```

where the MACVLAN value is the Nth network interface (alphabetically ordered), that lacks an assigned IP address.

- Assign 192.168.1.2 to the container in the start.config

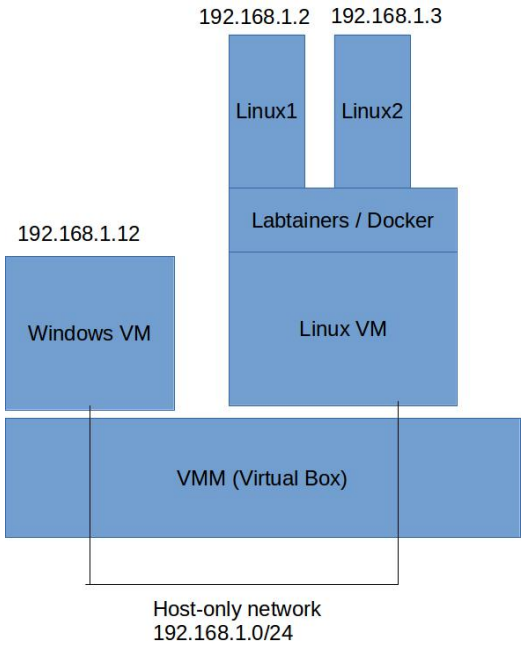

Figure 2: Networking with external hosts.

Also see the description of Multi-user labs in 9.

8 Building, Maintaining and Publishing Labs

Typically, when a Labtainer is started, the container's associated Docker images are pulled from the Docker Hub if they are not already local on the Linux host. When building and editing labs, the designer desires to run images reflecting recent changes that have been made. The framework includes logic to identify dependencies within containers whose image content has changed, and it will rebuild those images, (using the Docker build command). The framework will only rebuild those images that have changed. The designer can force the rebuild of all images within a lab by appending the "-f" switch to the end of the "rebuild.py" command. That switch is not intended for routine use because it wastes time and masks errors in our dependency logic.

If you build a new Labtainer exerciese, the container images will not be on the Docker Hub unless you put them there. If they are not on the hub, each student's computer will rebuild your lab when they go to run it. While this is fully functional, the build time may distract from performance of the lab. If you create your own public repository on the Docker Hub (https://hub.docker.com/), you can populate that with your lab(s) by setting the "REGISTRY_ID" value in the start.config file for the lab(s). You would then use the distrib/publish.py script to build, tag and push your lab to your registry.

8.1 NPS Development Operations

When building lab images at NPS, please set the LABTAINER_NPS environment variable to "YES", e.g.,

```
export LABTAINER_NPS=YES
```

This will force packages to be retrieved from the local NPS mirrors (centosmirror.uc.nps.edu or ubuntumirror.uc.nps.edu). Refer to section 8.4 for additional information.

Labs must be checked into the local Git repository in order to be distributed. After creating and testing a new lab, use the scripts/designer/bin/cleanlab4svn.py script to remove temporary files that do not belong in svn. Use the publish.py script (described above) to publish the lab containers. The distrib/mkdist.sh script is used by NPS to create the distribution tar file. This script relies on your local Git repository as the source to the Labtainer scripts and labs. Use the mk-devel-distrib.sh script to publish the developer configuration of the tar file.

The mkdist.sh and mk-devel-distrib.sh scripts include "myshare" variables that define a path to a directory shared with the development VM's host. The scripts will place the resulting tar files in this directory. You must then manually transfer the updated tar files (including the `labtainer_pdf.zip` file) to the Liferay server at

```
davs://my.nps.edu/webdav/c3o-staging/document_library/labtainers
```

After transfering the files, use the Liferay "Publish to Live" function to make the files available on the Labtainers website (which is also where they are pulled from when a student runs update-labtainer.sh).

Be sure to push your Git repository updates to the GitHub master.

The distrib/publish.py script is used to rebuild and publish individual labs, or optionally all of the Labtainer exercises managed by NPS. The publish.py (without the `-l` option) script will only rebuild labs that have changed. After pushing a new lab container image to the Docker hub, the script deletes the image from the local system. The intent is to ensure that future testing of the lab is done on the authoritative copy, i.e., from the hub.

Labtainer base images are built and published from the scripts/designer/bin directory. Prior to publishing baseline images, it is suggested that all local images be purged from the development machine, e,g.,

```
/trunk/setup_scripts/destroy-docker.sh
```

This will ensure the new baseline images to not incorporate layer remnants.

All new images should be first built and pushed onto the test registry, i.e., using the `./publish_image.sh <image> -t`

Framework modifications made to support changed or new functions within container images must be evaluated with respect to their impact on compatibility. If a new lab image requires an updated Labtainers framework, then the "framework_version" must be incremented within the bin/labutils.py script **before** the image is built and published. This will prompt users to run update-labtainer.sh prior to running any newer lab image. Also insure that these lines are present in the container dockerfile:

```
ARG version
LABEL version=$version
```

And, be sure to publish the revised framework before publishing the revised lab(s).

8.2 Alternate registry for testing

If the environment variable `TEST_REGISTRY`, is set to YES, labs to be pulled and pushed into an alternate registry defined in the trunk/config/labtainer.config file test_registry entry entry. Also, the `build_lab.py`, `labtainer`, and `publish.sh` scripts include `-t` flags to force the system to reference the test registry instead of the Docker Hub. It is easy to set up a registry (it is a container!), `https://docs.docker.com/registry/deploying/` Use the `trunk/setup_scripts/prep-testregistry.sh` script to prepare a test system to use a test registry.

8.3 Reuse of large file sets

The use of "sys_tar" and "home_tar" described in 4.3.1 facilitates sharing of common baselines of large or numerous files. New labs can incorporate tar files from existing labs through the use of "external-manifest" files, (see the xsite/victim/home_tar as an example). The syntax of the external-manifest is shown below, and it may contain multiple entries, one per line:

```
lab:container
```

Where "lab" is the name of the lab, and "container" is the name of the container whose tar file is to be included.

The framework will include content of tar archives referenced within these files when creating an archive for the new lab. This allows the sys_tar to include lab-specific files as well as files from other labs. Designers should avoid adding duplicate tar files to the SVN repository. This will avoid duplication of the files when a new distribution is created.

8.4 Package sources for apt and yum

Labtainer base images include configuration files to use local NPS mirrors when creating derivative images. The original apt or yum sources are restored to an image if it is built without an environment variable of `LABTAINER_NPS=YES` The original sources are also restored when any container is first run. See the baseline Labtainer Dockerfiles in `trunk/scripts/designer/base_dockerfiles` to understand how the sources files are manipulated.

The `apt_source` entry in the trunk/config/labtainer.config file will set the `$apt_source` environment variable in a Dockerfile, and this can be used by lab designers to force image builds to use alternate sources. By default, the value of the variable is "archive.ubuntu.com". This hostname can be overridden via the trunk/config/labtainer.config file apt_source entry, and having the following in your Dockerfile:

```
RUN sed -i s/archive.ubuntu.com/$apt_source/ /etc/apt/sources.list
```

8.5 Locale settings

The locale settings, (e.g., used when interpreting character encodings) are set to en_US.utf-8 as can be seen in

```
trunk/scripts/labdesigner/base_dockerfiles/Dockerfile.labtainer.base
```

Similar Dockerfile entries in new or existing labs can provide alternate locale settings.

8.6 Lab versions

Substantive changes to an existing published lab should be made in a new named lab. A *substantive* change is defined as one that would break any existing installation in a manner that could not be corrected with a framework update. Issues with compatibility between two lab versions is often due to there being lab-specific files on the framework, (i.e., from where the lab is run) as well as within the Docker images that make up the lab. When a newer version of a lab image is published, it must be able to work with existing installations. If that requires an update to the framework, then that update cannot break any existing labs present in that installation, i.e., labs that have already been started.

For example, **never** change container names for existing labs. If such a change is needed, create a new lab, and assign version numbers to it and the old lab.

Lab version numbers are kept in the optional `labs/[lab]/config/version` file. There is no need to have such a file until there are two or more versions of the same lab. (Note if you want two versions of a given lab to be runnable and to appear in the list of labs, then they are not versions of the same lab. They are different labs.) The format of a lab version file is:

```
lab-base version
```

where `lab-base` is a name to associate with the multiple versions. It can be anything and does not appear at the user interface. The `version` is an integer.

To create a new version of a lab:

- Create a new lab using `new_lab_setup.py` (perhaps with the clone option).
- Create a version file for the old lab (if it does not already have one).
- Create a version file for the new lab, giving it a version numerically greater than the old version.
- Add the old lab name to the trunk/publish/skip-labs list.

When the user types the `labtainer` command with no arguments, the list will then only include the latest version of that lab. An exception is if the old lab already has been run in this installation, in which case both lab versions will display.

9 Multi-user Labtainers

Labtainer exercises can support multiple concurrent users, such as students collaborating or competing on a shared set of networked components. A multi-user lab can be operated in any one of three modes:

1. Dedicated to a single student, e.g., on a laptop or a VM allocated to the student from a VM farm.
2. Shared by multiple students, each running Labtainers on a per-student VM with shared componets running on separate Labtainers VM. This is illustrated in Figure 3
3. Shared by multiple students, each SSHing from a non-Labtainer VM into a per-student Labtainer computer on a single VM running Labtainers. This is illustrated in Figure 4

Both of the multiuser modes require a host-only network defined by the VMM. This network should be defined before it is allocated to any VM, and the DCHCP server on the host-only network should be disabled within the VMM.

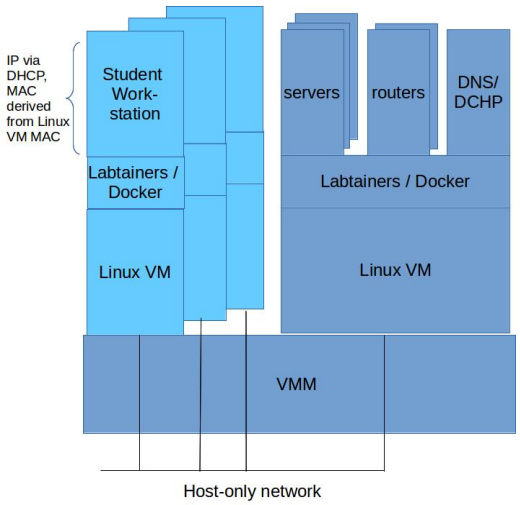

Figure 3: Multi-user Labtainers with multiple instances of Labtainers.

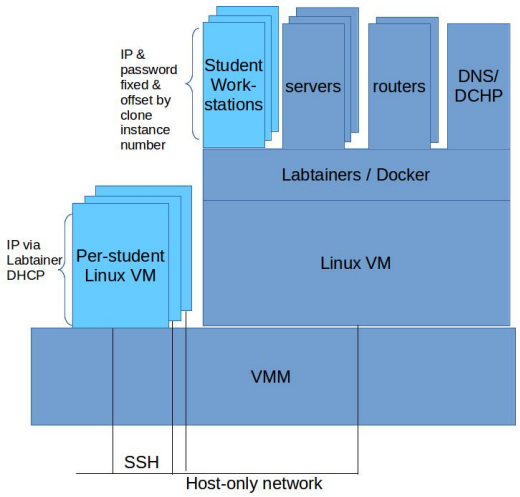

Figure 4: Multi-user Labtainers via SSH.

9.0.1 Multi-user Labtainers, one Labtainer VM per student

In this approach, each student is assumed to have been allocated an individual VM upon which Labtainers is installed. The student has access to that VM, e.g., via ssh or a vSphere client. Each student VM runs a single per-student workstation Labtainer component. The remaining containers, e.g., vulnerable servers, all run on a single VM, which we refer to herein as the "server VM". Provisioning a lab to run in this mode is summarized below.

- Allocate the host-only network to each Labtainers VM. Be sure to disable the IPv4 networking for this network on each Labtainers VM, and set the network interface to promisuous mode (within the Linux host as well). For example:

```
sudo ifconfig ethx 0.0.0.0
sudo ifconfig ethx promisc
```

- Start the lab on the server VM using start.py and the –server (-s) switch. This causes Labtainers to start each container in the lab that is not tagged as a "CLIENT".

- Students then start the lab on their individual VMs using start.py with the –workstation switch, which will cause the student VM to only start the container identified as the "CLIENT" in the start.config file.

When conformant labs (see 9.1) are started, the workstation containers use DHCP to optain IP addresses from a Labtainers DHCP server. The MAC address of the workstation container is derived from the MAC address of the Linux VM host-only network interface (to avoid duplicate MAC addresses on the host-only network).

9.0.2 Single Labtainers VM with multiple students

In this approach all the Labtainer containers will run on a single VM. Students have access to one or more other VMs hosted on the same VMM as the VM that hosts Labtainers. Students will SSH from these VMs into the container workstation allocated to the student via the host-only network. The ssh command may include the "-X" option to permit X11 forwarding, thus allowing students to run GUI-based applications on their workstation containers.

- Allocate the host-only network to the Labtainers VM. Be sure to disable the IPv4 networking for this network on the Labtainers VM, and set the network interface to promisuous mode (within the Linux host as well).
- Allocate the host-only network to each VM used by students to SSH into their Labtainers workstation. Configure the network on the VM to use DHCP (the host-only DHCP server should be disabled, the VM will get an IP from a Labtainer DHCP server.)
- Start the lab on the server VM using start.py and the –clone_count (-n) switch, specifying the quantity of per-student client containers to start.
- Students then ssh into their respective containers over the host-only network.

Conformant labs will assign each student workstation component an IP address in a sequence, starting from a fixed value. These IP addresses are allocated to the students.

9.1 Creating conformant multi-user labs

The following suggestions are intended to yield labs that can be started in any of the three operating modes.

- The lab should include a "client subnet" via which multiple VMs will communicate.
- Within the `start.config` file, identify this subnet as either a MACVLAN or a MACVLAN_EXT. The MACVLAN_EXT option will create a MACVLAN for this interface regardless of what mode the lab is started in, and is only intended for use if the lab includes an external host as described in 7.2.
- Itentify the client component within the `start.config` file using:

```
CLIENT YES
```

- Define the client component network address for the client subnet using the `+CLONE_MAC` option, e.g.,

```
LAN 192.168.1.10+CLONE_MAC
```

When run in single user mode, the `+CLONE_MAC` suffix is ignored. When run with multiple Labtainer instances, the last four bytes of the network MAC address for each client is cloned from the network interface tied to the MACVLAN. When all multi-user Labtainer workstations run on a single VM, then the IP address is incremented by one less than the clone instance number.

- Include a dhcp server as one of your containers, e.g., per the dhcp-client lab. The labtainer.network base includes the dnsmasq service, which includes a DHCP server. Reconfigure dnsmasq to start the DHCP service, e.g., in your fixlocal.sh script.

- Edit the dhcp container's `_system/etc/dnsmasq.conf` file to include the range of DHCP addresses you wish to allocate to the clients. When multiple instances of Labtainers are run, then "client" is the per-student Labtainer workstation. When there is a single Labtainers VM, then "clients" are the VMs from which students SSH into their Labtainer workstations. An example dnasq.conf entry is:

  ```
  dhcp-range=192.168.1.10, 192.168.1.99, 12h
  ```

- Enable dhcp on the client workstation components by installing isc-dhcp-client (via the Dockerfile), and putting this in the workstation _system/etc/rc.local:

  ```
  /sbin/dhclient-labtainer eth0
  ```

  Note the dhclient-labtainer invokes the dhclient program and then manually sets the ip address. This is a workaround for a Docker limitation.

- Include an SSH server in the workstation container, e.g., by deriving it from the labtainer.network base. Include a `_system/etc/ssh/sshd_config` file for the workstation container that permits X11 forwarding (if desired), e.g., by copying the file from the kali-test lab.

- Password management (only has an effect in multiuser mode when all Labtainers components are on a single VM). Assuming you'd like to allocate each student a unique (insecure) password for purposes of further ensuring one student does not accidently ssh into some other student's workstation, put this in the workstation's _bin/fixlocal.sh file:

  ```
  newpwd=studentCLONE\_NUM
  user=$2
  /usr/bin/passwd $user <<EOF
  $1
  $newpwd
  $newpwd
  EOF
  ```

  Add, this in the labs/[your lab]/config/parameter.config

  ```
  PASSWD : CLONE_REPLACE : .local/bin/fixlocal.sh : CLONE_NUM : CLONE
  ```

10 Limitations

The labtainers framework limits labs to the Linux execution environment. However, a lab designer could prescribe the inclusion of a separate VM, e.g., a Windows system, and that VM could be networked with the Linux VM that hosts the Docker containers as described in 7.2. Future work would be necessary to include artifacts from the Windows system within the framework's automated assessment and parameterization.

The user does not see the /etc/fstab file. Only virtual file systems can be mounted (or those mounted when the container is created.)

Kernel logs do not appear in `/var/log/kern.log`. For logging events such as iptables, consider using ulogd and a "NFLOG" directive in place of a "LOG" directive. See the dmz-lab as an example.

The available Docker network drivers do not permit IP address overlap between virtual networks. For example, you cannot define two 192.168.1.0/24 LANs.

Student use of the shell directive "source" will cause stdin/stdout to not be captured.

Inquisitive students will see evidence of artifact collection. Home directories on containers includes a `.local` directory that includes Labtainer scripts that manage capturing and collection of artifacts, and that directory contains the stdin and stdout files generated by student actions. Additionally, when the student starts a process that will have stdin and stdout captured, the student will see extra processes within that process tree, e.g., the `tee` function that generates copies of those data streams. All of the containers share the Linux kernel with the Linux host. Changes to kernel configuration settings, e.g., enabling ASLR, will be visible across all of the containers.

11 Notes

11.1 Firefox

The labtainer.firefox image includes a /var/tmp/home.tar which is expanded into the user home directory when parameterize.sh is run. This tar includes a profile in .mozilla that avoids firefox starting with its welcome pages and privacy statements. The labtainer.firefox image includes a customized /usr/bin/firefox that starts the browser in a new instance so it does not share existing browsers. The `about:config` was altered to disabled insecure field warnings for the labs that do not use SSL connections to web servers. If you wish to assess places a browser has visited, e.g., use a pregrade.sh to extract sites from the firefox places.sqlite file, put `places.sqlite` into the lab's /_bin/noskip file.

See A.2 for information on avoiding firefox crashes when it is restarted in SimLab.

11.2 Wireshark

Wireshark will not run as root in Labtainer containers. The wireshark installion in the labtainer.wireshark image is configured to not require root to collect network packets:

When using the wireshark image, after the existing

```
RUN adduser $user_name sudo
```

add:

```
RUN adduser $user_name wireshark
```

11.3 Elgg

The xsite/vuln-site/myelgg.sql file needs to be loaded for elgg to run. First edit it to change xsslabelgg.com to your site name (two changes). Copy the sys_tar/var/www/xsslabelgg.com/elgg to your new lab. Note the elgg/views/default/output files have been modified to permit cross site scripting.

11.4 Host OS dependencies

On rare occations, performance of a lab may depend on the host Linux OS. An example is some kernel tuning parameters viewed and set via sysctl are not visible within containers on eariler versions of Ubuntu. If your lab has such OS dependencies, you can check the OS and warn the student/instructor via a script named "hostSystemCheck.py" placed withe the _bin directory of any of the lab's containers. This script shall return the value '0' if dependencies are met, and '1' if dependencies are not met. In the latter case, the startup.py will prompt the use to continue or abort. Your script should explain the situation to the student. An example of such a script is in the labs/tcpip/server/_bin directory.

11.5 Login Prompts

See the centos-log lab for an example of a lab that prompts users to login to the virtual terminal. In particular, you will need the `bin/student_startup.sh` script, and the `_system/sbin/login` program and the `_system/etc/login.defs` and `securetty` files.

11.6 Networking Notes

11.6.1 SSH

The labtainer.network baseline Dockerfile includes the following:

```
ADD system/var/run/sshd /var/run/sshd
RUN sudo chmod 0755 /var/run/sshd
```

11.6.2 X11 over SSH

The scripts/designer/system/etc/ssh/sshd_conf allows X11 tunneling over ssh, e.g., from a remote VM connected to the same host-only lan as a container running the GUI application. Use `ssh -X container_ip` to enable X11 tunneling in the ssh session.

11.6.3 Traffic mirroring

Send copies of traffic from one ethernet port to another using the iptables TEE operation, e.g.,

```
iptables -t mangle -A PREROUTING -i eth1 -j TEE --gateway 172.16.3.1
```

will send copies of all incoming traffic on eth1 to the component with address 172.16.3.1. Note that gateway must be a next hop, or you will have to configure the nexthop to forward it further. This is useful for IDS labs, e.g., snort. Mirroring all incoming traffic into a component will let you reconstruct TCP sessions within that component. Mirroring output from components is not always reliable. Besides potential for duplicate traffic, Docker networks seem to sometimes gratuitously replace destination addresses with those of the Docker network gateway, i.e., the gateway to the host.

11.6.4 DNS

Install bind9 in Dockerfile. Add zone files to /etc/bind and db files to ../_system/var/cache/bind/. Add reference to the /etc/bind/named.conf.local as seen in local-dns/_bin/fixlocal.sh

11.6.5 Overriding Docker routing and DNS

Realistic network topologies require components to have /etc/resolv.conf and routing table entries that do not depend on Docker gateways and related magic. However, at some point you may want components to be able to reach the outside world. If you've fiddled resolv.conf and routing, you likely broke the default Docker method for doing this. One solution is to define an *isp* component that has a default gateway and resolv.conf as Docker defines them. Then route all traffic and DNS queries to that (making use of dnsmasq and your own resolv.conf entries). Note you will also have to set up your own NAT on that ISP component. See the dmz-example lab ISP component .local/bin/fixlocal.sh as a worked example of a simple NAT setup.

As a worked example, the dmz-example lab components (other than the ISP), typically use the .local/bin/fixlocal.sh script to delete the Docker-generated route:

```
sudo route del -host 172.17.0.1
```

And the fixlocal.sh also replaces the resolv.conf entry with either a local DNS component, or a gateway running the dnsmasq utility. The /etc/rc.local script generally sets the default gateway, and configures iptables.

11.7 User management and sudo

The Dockerfile should make the initial user, i.e., the user named in the start.config file, a member of sudoers. Otherwise, the fixlocal.sh script will not be unable to modify the environment. If desired, that user can be removed from sudoers at the end of the fixlocal.sh script.

Only the initial user (and that user's actions taken as root) are monitored. Additional users can be added, e.g., in the Dockerfile, but their actions are not monitored or recorded in artifacts.

11.8 Suggestions for Developers

The result and goals configuration files can be revised and tested within a running instructor container. This saves time because you do not need to rebuild the container for each iteration of the development of configuration files. However, be sure to scp the configuration files from the container to your host Linux system.

Most result and goal assessment can occur once you have generated a suitable sample of expected student artifacts. In other words, adding new goal does not typically require that you go back and re-perform student actions. Exceptions to this are:

Use `TERMINAL_GROUPS` in the start.config file to organize terminals if you have more than a few. Otherwise the student will spend time trying to find each terminal.

1. Adding new system commands to a "treataslocal" file;

2. Identifying new system files to be parsed as results. For example, results in a log file will not be collected unless that log file has been named in the results.config file.

11.9 Container isolation

Docker provides namespace isolation between different containers, and between the containers and the host platform. Note however, that all containers and the host share the same operating system kernel. Some kernel configuration changes will affect all containers and the host. For example, use of sysctl to modify Address Space Layout Randomization (ASLR) will effect all containers and the effects will persist in the host after the containers are stopped. However, some tuning parameters such as net.ipv4.ip_forward are isolated, i.e., local to the container. These do get reset in ways that are hard to predict, so it is suggested that sysctl tuning be done in rc.local scripts so that they happen on each boot.

Note also, that the Docker group (in which containers execute) is root equivalent, and thus a hostile container can do damage to the Linux host.

11.10 Student self assessment

The `checkwork` command allows students to assess their own work against the criteria used by instructors for automated assessment of lab performance. This can be disabled on deployment-wide basis using the `CHECKWORK no` directive in the `config/labtainers.config` file. Of course this assumes you have separately provided access control over that file, e.g., through use of a custom VM appliance.

A SimLab for testing labs

SimLab is a testing tool used to simulate a user performing a lab. It utilizes the xdotool utility to generate keyboard input and window selections. The SimLab tool is driven by a sequence of directives stored in a file at this location:

`labtainer/simlab/<labname>/simthis.txt`

Note that simlab files are not in the svn trunk or in the github repository. These files essentially contain lab solutions, and thus should not be openly published. [18]

With SimLab, you can fully automate the performance of a lab, including the use of GUIs. This facilitates regression testing, and the initial development of labs – particularly the debugging of automated assessment.

Full automation of regression testing is achieved using the `smoketest.py` utlity described below in A.3

A.1 SimLab Directives

Directives within a `simthis.txt` file name windows to select, (i.e., gain focus), and keystrokes to generate as described in the list below. The SimLab utility includes limited synchronization features to pause the input stream. These currently include a directive to wait until some named process has completed execution; and a directive to wait until network connections with a given host have terminated.

The SimLab directives are as follows:

- **window** <text> – Selects the window having a title that contains. Note that tabs within windows are selected by first selecting the window, and then use `key "ctrl+Next"` to tab over to the desired terminal tab. the given text. Will timeout and fail after 20 seconds.
- **window_wait** <text> – Like window, but no timeout. Intended for use when the xterm title is changed by a program.
- **type_line** <text> – Types the given text.
- **key** <keysym> – Performs a keypress for the given X11 keysim, see `http://xahlee.info/linux/linux_show_keycode_keysym.html` and `https://www.in-ulm.de/~mascheck/X11/keysyms.txt`
- **rep_key** <count> <keysym> – Repeats a keypress for the given X11 keysim <count> times.
- **sleep** <seconds> – Sleeps for the given number of seconds.
- **wait_proc** <text> – Delays until a `ps au` <`text`> returns nothing. Intended for use to wait for a command to complete. This runs on the Linux host, so do not be vauge, or it may never return.
- **wait_net** <container>:<text> – Delays until network connections to a given remote host have terminated. The given <text> is searched for as a substring within the host name ouput from a `netstat` command run on the given container.

[18] If you require simlab files for existing labs, contact me and try to convince me you actually need them (mfthomps@nps.edu).

- **type_file** <file name> – Reads and types each line in the named file. Blank lines will cause a 2 second sleep.
- **command_file** <file name> – Intended for use in issuing a series of commands from the shell. This reads and types each line in the named file. A `wait_proc` function is then automatically performed on the line.
- **key_file** <file name> – Reads each line in the named file, and performs a keypress. The lines should contain X11 keysims. Blank lines cause a 2 second sleep.
- **replace_file** <source file> <container>:<dest file> – Copies content of a source file on the Linux host relative the simlab directory, to a destination path on the named selected container.
- **add_file** <source file> <dest file> – Will append text from the source file to the end of the destination file. The destination file will be accessed from the currently selected virtual terminal. This uses a simple VI scheme to append text, and thus assumes the window and cwd are as needed.
- **include** <file> Reads the named file and treats each line as a SimLab directive, and then continues processing the next directive in the source file. This is similar to the C include directive.
- **type_function** <command> – Will execute the given command, read stdout from the command and then type that.

A.2 SimLab spplication notes

Most GUI's have shortcut keys that can be used to automate their inclusion in a lab.

Firefox is brittle when it restarts. See the `fixfirefox.txt` SimLab script for the snort lab for an example of avoiding errors when Firefox restarts.

A.3 Regression testing with smoketest.py

The `smoketest.py` utility automates regression testing of labs. It will automatically:

- Start a lab
- Use SimLab to perform the lab
- Stop the lab
- Use `gradelab` to assess the lab
- Compare the results of gradelab to those stored in the directory at:

  `labtainer/simlab/<labname>/expected/`

If `smoketest.py` is started with no parameters, it will iterate through each lab in the labs directory. The that lab lacks `simthis.txt` file, then the lab is simply started and stopped (hence the tool's name). The tool will stop upon encountering the first error.