This document was created by United States Government employees at The Center for the Information Systems Studies and Research (CISR) at the Naval Postgraduate School NPS. Please note that within the United States, copyright protection is not available for any works created by United States Government employees, pursuant to Title 17 United States Code Section 105. This document is in the public domain and is not subject to copyright.
Contents

1 Introduction ................................. 2
   1.1 Intended use ................................ 2
   1.2 Benefits of Labtainers ..................... 2

2 Overview of the Student Environment and Workflow 2

3 Obtaining the Labtainer Development Kit 3

4 Defining New Labs .......................... 3
   4.1 Testing the new lab ......................... 4

5 Defining the lab execution environment .......... 5
   5.1 Container Isolation ......................... 5
   5.2 Naming Containers .......................... 5
   5.3 Container definitions in start.config ... 6
   5.4 Lab-specific files in the student’s home directory ... 7
   5.5 Lab-specific system files .................. 8
   5.6 System services ............................ 8
   5.7 Instructions for Students .................. 8
   5.8 Running programs in Virtual Terminals ... 8
   5.9 Final lab environment fixup ............... 8
   5.10 Automatic copying files from containers to the host ... 9

6 Parameterizing a lab ........................... 9

7 Automated assessment of student labs ............ 11
   7.1 Capturing information about the environment ... 13
   7.2 Assessing the student results ............... 13
   7.3 Example Simple Parameterization and Grading ...... 15

8 Networking .................................... 16

9 Building and Publishing Labs .................. 16

10 Limitations .................................. 16
1 Introduction

This manual is intended for use by lab designers wanting to create or adapt cyber security labs to use the Docker-based lab framework known as "Labtainers".

1.1 Intended use

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 lab may include multiple networked components, all running locally on a student’s computer.

1.2 Benefits of Labtainers

Deploying cyber security 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 containers 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 another 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.

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 environments, primarily via bash shell commands. In general, the Labtainer framework implementation is not visible to the student, and the Linux environment as seen by the student is not augmented to support the framework. The Linux execution environments presented by Docker containers do not readily include GUIs. As a result, Labtainer should not be used for exercises requiring GUIs.

Labtainers are intended for use on individual student computers. 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 "host", can be any distribution and version which supports
Dockers, and it must have Dockers installed. In addition to installing Dockers on their Linux host, the student must obtain and expand a tarball, which contains the Labtainer workspace utilities. (This tarball may someday be replaced by standard Linux distribution packages, e.g., Debian and/or RPM packages.) 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. In its current implementation, a single base image (about 500MB) is retrieved from the Docker registry, and the remainder of the image is built on the student’s computer, i.e., by fetching packages. This is transparent to the student, other than waiting for downloads and package installation.

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

Many cyber security 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 does not prescribe mechanisms for managing such reporting. If a given lab exercise was not designed to support automated assessment, then zip files from the students need not be collected. On the other hand, instructors may still wish to collect the zip files to manually review selected student activity, e.g., student-developed programs, output files, or even bash history records.

## 3 Obtaining the Labtainer Development Kit

Installation of Labtainers is described in 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 Docker package. Note Labtainers will work with any Linux distribution that supports Dockers. If you already have Dockers installed on a Linux system, reference the Student Guide for other dependencies. Future releases may include a separate development kit, but for now there is a single Labtainers distribution that contains the entire framework.

## 4 Defining New Labs

The most challenging and critical part of designing a new cyber security 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 burdens you are placing on students and instructors. The framework does not require lab designers to program or create scripts. The lab
designer primarily interacts with the framework by editing configuration files which affect the student’s execution environment and the optional automated assessment of student activity.

The examples in this guide assume you have defined the LABTAINER_DIR environment variable. Set the LABTAINER_DIR environment variable to the top of the svn repo, e.g.,

```bash
export LABTAINER_DIR=/home/mike/labtainer/trunk
```

It is suggested that you add the LABTAINER_DIR definition in your .bashrc.

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 named used by students when starting the lab. It must be all lower case and not contain spaces.

```bash
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.

```bash
$LABTAINER_DIR/scripts/designer/bin/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. The result is a new labtainer lab that can be run. While this new lab will initially only present you with a bash shell to an empty directory, it is worth testing the lab to understand the workflow.

### 4.1 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 set the LABTAINER_DIR environment variable if you had not added it to your .bashrc. Then:

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

Then start the container using the:

```bash
./redo.py [labname]
```

command, where labname is the name of the lab you just created. The very first time you run this, it may take a bit of time because it fetches the base Labtainer Docker image from the Docker registry. Subsequent builds should be faster because your local Docker system will cache portions of the build.

The redo.py command will remove and recreate the container each time the script is run. And it will rebuild the container image if any of its configuration information has changed. This is often necessary when building and testing new labs, to ensure the new environment does not contain artifacts from previous runs.

Note the "redo.py" command is not intended for use by students, they would use the "start.py" command.

Stop the containers with

---

1The build process may generate warnings in red text, some of which are expected. These include an un referenced "user" variable and the lack of apt-utils if apt-get is used to install packages in Dockerfiles.
Note that when you stop the container, 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 redo.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 container with stop.py, you will see the stdin and stdout results of the program within the saved zip file.

Note how 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 Defining the lab execution environment

A given lab typically requires some set of software packages, and some system configuration, e.g., network settings. The framework captures most configuration details within a standard Dockerfile. A default Labtainer-specific Dockerfiles is placed in the new lab’s "dockerfiles" directory when the new lab is created. We use standard Docker file syntax, which is described at https://docs.docker.com/engine/reference/builder/.

Lab designers should reference that Docker documentation for the syntax and semantics of these files. 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.

A given lab 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.

5.1 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. 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.

5.2 Naming Containers

If a lab is to include only one container, you can skip ahead to the subsection titled Lab-specific files in the student’s home directory.

You must assign a name to each container within the new lab. Each new lab starts with a single container, whose name matches the lab name. You are free to change that name. The
names of containers should reflect their role in the lab, e.g., "client" and "server". Once you
have picked container names, you must create Dockerfiles and update the

$LABTAINER_DIR/labs/[labname]/config/start.config file.

Each container must have its own Dockerfile within the

$LABTAINER_DIR/labs/[labname]/dockerfiles
directory. The naming convention for dockerfiles is

Dockerfile.[labname].[container_name].[role]

where role is either "student" or "instructor". The framework will use the student Dockerfile
to create the instructor container unless a distinct instructor file is present (e.g., if the designer
wishes the instructor to have unique software packages.) The system automatically creates a
student Dockerfile when a new lab is created. You are responsible for creating (copying to)
additional Dockerfiles for other containers, and changing the name of the initial Dockerfile if
its container name changes.

The first line of each Dockerfile identifies the base 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

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

5.3 Container definitions in start.config

Most single container labs can use the automatically generated start.config file without
modification. Labs consisting of multiple containers must modify the start.config file. The
following describes the major sections of that configuration file.

• GLOBAL_SETTINGS – Beneath this keyword, the following values must be defined:
  
  – GRADE_CONTAINER [container name] – All lab containers are available to the
    instructor while assessing student labs. This setting identifies which of the lab con-
    tainers will host automated grading functions.
  – 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 parameteri-
    zation of individual student labs.
  – REGISTRY [registry] – The id of the Docker Hub registry that is to contain the lab
    images. See ?? for details on the use of this keyword.

• 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

- CONTAINER [container name] – One of these sections is required for each container in the lab. 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.
  - 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. If the user is not ”ubuntu”, then the Dockerfile must include an entry to add the user to the container. (See the nmap-ssh lab mycomputer dockerfile as an example).
  - 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.

A simple example of a two-container lab can be found in

\$LABTAINER_DIR/labs/telnetlab

### 5.4 Lab-specific files in the student’s home directory

Files that are to reside in the student’s $HOME directory are placed in the new lab container directory. For example, if a lab includes a source code file, that should be created in the lab container directory, and it 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]

Note the name of 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. The Dockerfile should not include any other ADD commands to copy files to the HOME directory.
5.5 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. The Dockerfile should not include any other ADD commands to copy system files.

5.6 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. The default Labtainer Dockerfiles include an ENTRYPOINT command that launches a faux_init script that starts rsyslog, (so that system logs appear in /var/log), and runs rc.local. The 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 faux_init script can be extended (or replaced) as needed by the lab. The example telnetlab illustrates the use of this script to provide network services.

5.7 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. 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.

5.8 Running programs in Virtual Terminals

The "XTERM" section in the

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

5.9 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 after they have been parameterized, (as described below). 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
5.10 Automatic copying files from containers to the host

Some labs require the use of tools that are not available within containers. For example, Wireshark cannot run in a container because it requires a GUI. In such cases, the student can be directed to run the tool from the host, however Labtainers does not support automated assessment of the results. However, the files accessed by such tools can be parameterized by Labtainers, and then copied to the host. Such 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.

6 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. 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 hexadecimal 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).

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.

<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
(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

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 concatenatated 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.
<string> - 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".

The parameter_id fields may be referenced during the automated grading function, described
below.
7 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 that labs include automated assessment, e.g., the "results" of a lab may consist entirely of a written report submitted by the student.

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.

The framework’s automated assessment functions generally assume the student will interact with one or more programs or system utilities. 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. These timestamped file sets, and everything else 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 CLE), and ingested into the instructor’s system where lab assessment occurs.

The 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 selected system programs will be captured if the program names appear in the `treataslocal` file at

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

System utilities should include their full path, e.g., "/usr/bin/nmap". Non-system programs can be excluded from stdin/stdout capturing by including their names in a "ignorelocal" file in that same directory.

{sectionIdentify Lab-specific Artifacts The automated assessment functions encourage labs to be organized into a set of distinct "goals". For each goal, the lab designer should identify specific fields within stdin and/or stdout that could be compared to "expected" values. These lab-specific artifacts are identified within the "instr_config/results.config file". Artifacts are identified in terms of:

1. the program that was invoked;
2. whether the artifact is in stdin or stdout
3. the line containing the artifact, and a token within that line.
4. ad-hoc properties, such as the quantity of lines in the stdin file.

Each identified artifact is given a symbolic name. A named artifact is referred to herein as a `result`, which is then referenced in the goals.config file to assess whether it is an expected value.

Consider the labs/formatstring/instr_config/results.config file. The first non-comment line defines a result having the symbolic name "crashStringCanary". This result is found by looking at stdout from the "vul_prog" program, finding the first line that starts with: "*** stack smashing detected". The result is assigned the value of the third space-delimited token in that line.

Entries within the results.config file each have the following format:
<nametag> = <file_id> : <field_type> : <field_id> : <line_type> : <line_id>

where:

- **nametag** -- 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] | file_path

  where <prog> is a program or utility name whose stdin and stdout 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.

- **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 parenthesis
  - **SLASH** -- The desired value is contained within slashes, e.g., /foo/
  - **LINE_COUNT** -- The quantity of lines in the file. Remaining fields are ignored.
  - **CONTAINS** -- The value of nametag will be set to true if the file contains the string represented in field_id.
  - **SEARCH** -- The value nametag will be set to the search resulting from the given field_id, which is treated as an expression having the syntax of python's parse.search function. E.g., "frame.number=={:d}" would yeild the frame number.

- **field_id** -- An integer identifying the nth occurrence of the field type. Alternately may be "LAST" for the last occurrence 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.

- **line_type** -- Identifies how the line is to be identified, values include:
  
  - **LINE** -- The line_id will be 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 will be a string. This names the first occurrence of a line that starts with this string.
  - **HAVESTRING** -- The line_id will be a string. This names the
first occurrence of a line that contains the string.

NEXT_STARTSWITH -- the line_id will be a string. This names the line preceding the first occurrence of a line that starts with this string.

line_id - See line_type above.

7.1 Capturing information about the environment

Some labs require the student to alter system configuration settings, e.g., using the sysctl command to effect ASLR. A checklocal.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 checklocal.sh script is recorded at the beginning of the stdout artifact.

7.2 Assessing the student 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 assess whether the student obtained expected results. Each lab goal defined in the goals.config file will evaluate to TRUE or FALSE, with TRUE reflecting that the student met the defined goal. Once evaluated, a goal may determine the state 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 a student grade.

As noted earlier, 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, the assessment determines if the student obtained expected results during any invocation of a program or system utility. In those cases where the student is required to obtain the expected results during the last invocation of a program, the matchlast goal type may be specified as described below.

To facilitate grading multiple attempts or explorations of a lab exercise, the framework associates timestamps with the results of processing the results.config file. A single timestamp will include results from a stdin file and a stdout file. In general, there will be a distinct, timestamped set of results for each occurrence of a student invoking a targeted program.

The following syntax defines each goal or subgoal within the goals.config file. While the syntax may appear complex, most goals can be expressed simply as can be seen in the labtainer examples distributed with the framework.

<goal_id> = <type> : [ <operator> : <resulttag> : <answertag> | <boolean_expression> | goal1 : goal2 | <resulttag> | count_greater : value : subgoal_list ]

Where:

<goal_id> - An identifier 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.

'matchlast' - only results from the latest timestamped set are evaluated.

'matchacross' - The answertag must name a result, and that result must occur in a timestamped set that differs from the set in which the resulttag occurs. Note: 'matchacross' cannot be used within the boolean expression defined below.

'is_true' - The goal is true if the value of the resulttag is true, e.g., from a "CONTAINS" field type in the results.config

'is_false' - The goal is true if the value of the resulttag is not true, e.g., from a "CONTAINS" field type in the results.config

'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 parenthesis for precedence. The goal_id’s must be from goals defined earlier in the goals.config file. The goal evaluates to TRUE if the boolean 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.

'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* goal types. Evaluates to TRUE if any TRUE goal1 has a timestamp that is before than any TRUE goal2

'time_during' - Both goal1 and goal2 must be goal_ids from previous *matchany* goal types. Timestamps include a start and end time, reflecting when the program starts and when it terminates. Evaluates to TRUE if any TRUE goal1 has a start timestamp within the start and end times of any TRUE goal2.

'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

<operator> - the following operators evaluate to TRUE as described below:

'string_equal' - The strings derived from <answertag> and <resulttag>
are equal.

'\text{string\_diff}' - The strings derived from $\text{answertag}$ and $\text{resulttag}$ are not equal.

'\text{string\_start}' - The string derived from $\text{answertag}$ is at the start of the string derived from $\text{resulttag}$.

\text{example:} \quad \text{answertag value} = '\text{MySecret}'
\quad \text{resulttag value} = '\text{MySecretSauceIsSriracha}'

'\text{string\_end}' - The string derived from $\text{answertag}$ is at the end of the string derived from $\text{resulttag}$.

\text{example:} \quad \text{answertag value} = '\text{Sriracha}'
\quad \text{resulttag value} = '\text{EatMoreFoodWithSriracha}'

'\text{integer\_equal}' - Integers derived from $\text{answertag}$ and $\text{resulttag}$ are equal.

'\text{integer\_greater}' - The integer derived from $\text{answertag}$ is greater than that derived from $\text{resulttag}$.

'\text{integer\_lessthan}' - The integer derived from $\text{answertag}$ is less than that derived from $\text{resulttag}$.

'$\text{executable\_file}$' - If the type is 'execute' then $\text{operator}$ is a filename of an executable.

$\text{resulttag}$ -- 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. An alternate syntax is to add "result." as a prefix to the symbolic name.

$\text{answertag}$ -- Either a literal value (string, integer or hexadecimal), or a symbolic name defined in the results.config file or the parameters.config file:

\text{answer}=<\text{literal}> \quad -- \text{literal string, integer or hex value (leading with 0x), interpretation depending on the operator as described above.}
\text{result.}<\text{symbol}> \quad -- \text{symbol from the results.config file}
\text{parameter.}<\text{symbol}> \quad -- \text{symbol from the parameters.config file}
\text{parameter\_ascii.}<\text{symbol}> \quad -- \text{symbol from the parameters.config file}
\text{same as above, but the value parsed as an integer or hexadecimal 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.

7.3 Example Simple Parameterization and Grading

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 watermark some file on one of the containers. An example of this technique is seen in the "iptables" lab. Specifically, the student_watermark entry in results.config, the own_work entry in goals.config, the DIGESTFILE entry in the config/params.config and finally, the filetodi-
gest.txt file on the server. (Our to-do list includes building this feature into all labs so that designer need not separately add it.)

8 Networking

Most networking is simply a matter of defining networks and assigning them to containers as described in 5.3.

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 some instances, the lab designer will want to define a different default route for a container. Each container derived from the Labtainer network base includes a set_default_gw.sh script that can be added to the /etc/rc.local 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.

9 Building 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 ”redo.py” command.

If you build a new Labtainer lab, the container images will not be on the Docker Hub unless you put them there. In such a case, 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.

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. Future work would be necessary to include artifacts from the Windows system within the framework’s automated assessment and parameterization.

The process tree of the initial Linux process will not look like a typical Linux system init process. Within containers that have no services, the initial process, i.e., process ID 1, will be a bash shell. Containers having services and logging will have an initial process that is the script that launches the services, e.g., the faux_init script. However, other process tree’s will appear as they do in Linux system, and this includes inetd services.
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.