Labtainer Framework Development Guide

June 19, 2019

Contents

1 Introduction

This document is intended for use by developers who maintain the Labtainer framework. It does not address lab creation, which is covered in the *Labtainers Lab Designer User Guide*.

2 Developer Software Prerequisits

- Subversion
- Latex (texlive-full)

3 Getting Labtainers from Subversion

svn co https://tor.ern.nps.edu/svn/proj/labtainer Change directory to trunk/setup-scripts and run ./after-checkout.sh. The will build the PDF lab manuals so that you can reference the manuals while you test or otherwise reference existing labs. (Please follow the lab manual and report discrepancies!) It will also create any executables required by the framework. Then run ./pull-all.sh to pull all the baseline images (so that your running of existing labs is more akin to what students and instructors do so we can better test that).

4 Testing and Running Existing Labs

There are situations where you will run an existing lab, e.g., to test it, or to observe some example. When running labs, please refer to the lab manuals so that they get reivewed and tested by different people. Also, please first delete the lab using trunk/setup_scripts/removelab.sh to ensure that you are running the latest version of the published lab. If you find the lab to be broken, e.g., missing a file, please attempt to run "rebuild.py" on the lab. Report these findings to the lab author. And always run removelab.sh after you have run an existing lab via rebuild.py. Again, the goal is to force ourselves to run the distributed labs unless we have specific reasons to do otherwise.

5 Overview of Labtainer Elements

The Labtainer framework implementation is primarily python scripts. A number of the top level scripts share functions found in bin/labutils.py. The top level scripts are organized as follows:

- Student `labtainers` (start) and `stoplab` – In the labtainers-student/bin directory, these run on the Linux host and manage the pulling, starting and stopping of containers. They also coordinate collection of student artifacts.
- Student container scripts – In the labtainers-student/lab_bin directory, these execute on containers, e.g., to hook bash and parameterize containers.
- Instructor `gradelab` and `stopgrader`– Push student artifacts onto grader container and get assessment results.
- Instructor container scripts – perform grading functions.
- Developer building – rebuild.py in labtainers-student/bin and labtainers-instructor/bin.

- Publishing labs – labtainers/distrib/publish.py
- Base Labtainer images – /trunk/scripts/designer/bin, create and publish the base images.
- VM appliances – /trunk/host_scripts, update and publish VM appliances as OVA files for VirtualBox and VMWare.
- Regression testing of grading functions is performed by labtainer-instructor/regress.py. Expected results are stored in the labtainer/testsets directory.
- Regression testing of labs and grading combined: scripts in trunk/testsets/bin; data sets are not distributed, they are in labtainer/simlab/¡labname¿

6 Control Flow

Student scripts, e.g., start.py, run from the trunk/scripts/labtainer-student directory. That directory also contains the bin/labutils.py, which contains most of the framework functions.

When a student container is first started "docker exec" is used to run parameterize.sh on the container.

That script also invokes hookBash.sh, which adds the bash sdtin/stout capturing hook, and adds the startup.sh call into the .profile.

The startup.sh uses a lock to control which terminal displays the instructions or grading. In practice most instructions are now pdf files. The startup.sh invoked by student will source a student_startup.sh if present.

7 Automation and Distributions

The Labtainer framework is distributed via the c3o website as a tar file, or, optionally a VM applicance (both VMWare and VirtualBox). The Docker images are distributed via the Docker Hub.

The labtainer/distrib/mkdist.sh script runs on a Linux VM hosted on windows or Linux, and creates the distribution tar and copies it into a shared folder. The mk-devel.sh script makes the developers version of the tar. From that shared folder, the two tar files are copied to the

```
\\my.nps.edu@SSL\DavWWWRoot\webdav\c30-staging\document\_library"
```

and then "Publish to Live" is performed on the Liferay site.

Two prepackaged VMs are maintained: one for VirtualBox, and one for VMWare. Each include their respective guest additions. The VMs are maintained on a native Linux system using command line utilities, e.g., VBoxManage. The VMs are rigged to update labtainers, including a pull of baseline images, on each boot until the first lab is commenced. Scripts named "export*" are used to created the appliance files. The scripts re-import into test images, which must be manually tested. The WinSCP script pushes new applicance images to the CyberCIEGE download directory on the C3O web server. (Wine and WinSCP must be installed on the Linux host that manages the VMs.

New baseline images are created using scripts/designer/bin/create_all.sh. Note its comment about deleting all docker images first. When new baselines are created, use the labtainer-scripts on the native Linux system to update the VM appliances so they contain the latest baseline images. After the VM starts and updates the baseline images, use:

```
sudo dd if=/dev/zero of=/emptyfile bs=1M
sudo rm -fr /emptyfile
```

to zero unused space and then run

```
./poweroffVB.sh
./compact.sh
```

to compact the VM image. Then export it:

```
./exportVB.sh
```

This will create the appliance OVA image, and will create a test VM from that appliance. The test VM will start. Use that to run ad-hoc tests.

Do the same for vmware.

Then push the images to the web server

The appliances automatically update the baselines and the Labtainer scripts on boot, so there is only really advantage to doing this for baseline changes, since they take a while to download. After running the poweron/poweroff scripts, then run the exportVM.sh to

8 installation sizes

An initial install, including the base images, requires about 4GB. Installing a larger lab, e.g., snort, requires an additional 1GB. Running bufoverflow added 22M.

9 Notes

9.1 Race condition on checklocal.sh output

If an mynotify.py event causes an output from checklocal.py, that may conflict with concurrent output from checklocal.py resulting from some program/script running. In theory, the program/script should complete its run of checklocal before the program/script actually gets to access the file that triggers a mynotify watch. So, the latter's output to the timestamped file is appended. Further, the mynotify.py looks for an existing timestamped file, and if not found, looks for one from the previous second. This hack is an attempt to keep the outputs merged. It will fail if the access does not happen within a second of the program start. See the acl lab.

9.2 temporal logic considerations

When evaluating results from logfiles containing timestamps use FILE_TS or FILE_TS_REGEX to ensure you get timestamped values for only matching records. Reliance on goals.config to matchany can result in timestamped results that don't corrolate to the desired record.

9.3 parameterizing the start.config

A copy of the parameterized version of start.config is placed into labtainer-student/.tmp/¡lab¿/. This ensures that subsequent runs of the lab always have the same psuedo random values.

9.4 Packaging

The framework has not yet been adapted to use Linux package managers. Currently, scripts are run from a workspace directory and python paths are managed relatively between scripts.

9.5 Todo

Change smoke test to look for email in expected results and set that as the email before starting a lab. Validation should catch results.config naming of non-existent container.