Labtainers Cyber Exercises: Building and Deploying Fully Provisioned Cyber Labs that Run on a Laptop

(Workshop guide)

Mike Thompson
Cynthia Irvine

March 13, 2021

This workshop was supported by a grant from the National Science Foundation.

Contents

1 Introduction **2**

2 Prerequisites **2**

3 Labtainers overview **3**

4 Existing Labs (0:15-1:00) **3**

4.1 Perform the telnetlab 3

4.2 Assess telnet lab performance 4

4.3 Wireshark-intro lab 4

4.4 Individualizing labs: bufoverflow 5

5 Deployment options (1:00-1:10) **5**

6 Break (1:10-1:25) **5**

7 Building new labs (1:25-2:10) **5**

7.1 Start labedit 6

7.2 Review telnetlab implementation 6

7.3 Create a lab 6

7.3.1 Add software packages 7

7.3.2 Add files to a container 7

7.3.3 Modify /etc/hosts 8

7.4 Individualize the lab 8

7.4.1 Add a symbol to a lab file 8

7.4.2 Add a parameter directive 9

7.5 Add assessment 9

7.5.1 Treat as local 9

7.5.2 Name the artifact file 9

7.5.3 What from the file is of interest? 9

7.5.4 Define goals 10

7.5.5 Test assessment 10

7.6 SimLab 10

7.7 Assessment debugging with gradelab 11

7.8 Temporal assessment 11

8 Revisit lab examples (2:10-2:25) **11**

8.1 wireshark-intro parameter and assessment 11

8.2 bufoverflow parameters and assessment 12

9 IModules (2:25-2:40) **12**

10 Customized lab manuals (2:40-2:50) **13**

11 Conclusion and review (2:50-3:00) **13**

1 Introduction

This document is intended to guide participants of the SIGCSE Labtainers workshop. The workshop is scheduled to be roughly 3 hours in duration with a fifteen minute break and time for participants to interact.

2 Prerequisites

Prior to the start of the workshop, participants are expected to have downloaded and installed the Labtainers VM appliance from
`https://nps.edu/web/c3o/virtual-machine-images`.
This appliance requires that your system be running either VMWare or VirtualBox. **NOTE:** when downloading the appliance, use "save link as" to save the ova file, do not try to open the link with your browser or it might rename the file, leading to errors.

The VM appliance downloads are about 5GB, so please do not wait until the last minute before downloading and installing them.

After installing the VM appliance, start up the VM and let it complete its update process. (If you are using a Labtainers VM that was previously installed, run the `update-labtainer.sh` command.) This workshop includes creation of new labs, which requires installation extensions needed for lab creation. Using the command line of the terminal that was opened, type:

```
update-designer.sh
```

After the update completes, reboot the VM. You can then explore, and/or "Save" the VM and restore it at the start of the workshop.

Prerequisite Checklist

1. Install VMWare or VirtualBox
2. Download the Labtainer VM appliance and save to your system
3. Import the appliance into VMWare or VirtualBox
4. Boot the VM
5. Run `update-designer.sh`[1]
6. Reboot the VM

Once you have completed the prerequisites, you are ready to participate in the workshop. The remainder of this guide will be available prior to the workshop.

[1] If you are using an pre-existing Labtainers installation, first run update-labtainer.sh

3 Labtainers overview

Labtainers is a collection of cybersecurity lab exercises and a set of tools for creating, and deploying new labs, including general computer science exercises that run in a Linux environment. Labtainers uses Docker containers to provide fully configured, consistent lab environments that may include many networked computers.

In this context, *containers* are a mechanism for providing name-space isolation. This allows the processes within a container to reference a file system that is distinct and isolated from the host file system, and distinct from the file systems of other containers. The name-space isolation also allows containers to have independent virtual network connections and independent system configurations, e.g., files in `/etc/`. When suitably managed, containers look a lot like independent virtual machines. However, since each container shares the underlying kernel of the host, containers are much less resource-intensive than are virtual machines. This lets us spin up ten or more containers on a single laptop.

The Docker containers within each Labtainers lab are instrumented to capture artifacts created as students perform the exercises. These artifacts are automatically collected when the student completes the lab, and the student forwards the resulting zip file to the instructor. The instructor then uses Labtainers automated assessment to review student progress.

Lab exercises may be individualized to discourage sharing of solutions. For example, network addresses of computers within the lab may be randomized. Or the size of a buffer in a vulnerable program can vary by student. This can aid in the reuse of labs between different cohorts.

In this workshop, we will cover the use of some existing Labtainers exercises, including the student experience and automated assessment functions available to instructors. And we will build a lab exercise using the Latainers lab authoring tools. We will also look at how you can publish your new labs and share them with the community.

4 Existing Labs (0:15-1:00)

In this section of the workshop, we will explore existing Labtainer lab exercises to understand the student experience and automated assessment functions available to instructors.

For the benefit of those students who read directions, a link to the *Labtainer Student Guide* is on the VM desktop. You need not review that now. The desktop starts with an virtual terminal having labtainer-student as the working directory. This is where students start and stop labs. Type the following to view a list of available labs:

```
labtainer
```

This output is piped through `less`, so you can search or use q to quit when done viewing the list.

4.1 Perform the telnetlab

We'll first run the `telnetlab`. Enter the command:

```
labtainer telnetlab
```

Labtainers will automatically pull the necessary Docker images from Docker Hub. It then prompts you for an email address, which is used by Labtainers to individualize some of the labs and to identify each student in the automated assessment results. It will then provide a link to the lab manual. Right click on that link and open the lab manual and then press enter to start the lab.

Notice that two virtual terminals have opened, one labeled "client" and the other "server". These are two distinct computers connected via a virtual LAN. Use the `ps aux` command to observe the processes running on each.

This is a fairly simple lab. Perform the steps per the lab manual. When you are done, use `stoplab` to stop the lab.

Note the `labtainer` command as options, which can be viewed using the `-h` flag.

4.2 Assess telnet lab performance

When you stopped the telnetlab, Labtainers displayed the path to the `~/labtainer_xfer` directory containing a zip file that you would forward to your instructor. As the instructor, you have this very same `~/labtainer_xfer` directory, into which you would collect your student zip files. There is a shortcut to the `labtainer_xfer` directory on the VM desktop.

Assessment of student performance by an instructor occurs in the `labtainer-instructor` directory. Go there as follows from the `labtainer-student` directory[2]:

```
cd ../labtainer-instructor
```

The `gradelab` command performs student assessment functions. We will run it requesting the web-based interface:

```
gradelab telnetlab -w
```

The function displays a table of goals achieved by each student. Since only your results are currently in the `labtainer_xfer` directory, only your email address appears. The brief legend following the table summarizes each of the goals. Different labs have different amounts of automated assessment. At a minimum, the assessment provides some evidence that the student submitted a zip file generated during a Labtainers session started with their login ID. In some cases, the assessment provides evidence that the student solved a challenge, e.g., got a root prompt after subverting a service. Instructors may prefer to have students produce traditional lab reports and use automated assessment results to supplement reports and gain insight into hurdles encountered by the student.

Following the table, a link is displayed to `http://localhost:8008`. Right click on that and open the link. The resulting web pages allow the instructor to delve into the artifacts generated by each student. Click on the link named `Table of Student Goals`. Then click on your email address in the table, which will produce a web page allowing access to raw artifacts.

The timestamp table lets you view timestamped copies of standard input and standard output resulting from running selected programs, in this case, the telnet client.

Down toward the bottom you'll find links to bash history files for each component.

Run `stoplab` from the `labtainer-instructor` terminal to stop the grader (which is a Docker container.)

4.3 Wireshark-intro lab

Back at your `labtainer-student` directory, start the wireshark-intro lab:

```
labtainer wireshark-intro
```

Open the lab guide and perform the lab, which should only take five minutes. Alternately you can watch me perform the lab.

This is an example of including a GUI application in a lab exercise.

This lab is individualized by trimming a random number of packets from the start of a PCAP file and adjusting the timestamps within the file. The student is directed to locate a specific packet, and parameterization ensures the packet number and times seen by different students will vary.

[2]Consider using ctl-shift-tab to open a terminal tab so you can leave the original terminal in the labtainer-student directory

4.4 Individualizing labs: bufoverflow

Recall while performing the telnet lab, you displayed the content of a file on the server, and that file contained a hash value generated in part from the email address. And in the wireshark-intro lab, the PCAP file was individualized for each student. We'll now look at a different individualized lab. From the `labtainer-student` directory, start the `bufoverflow` lab:

```
labtainer bufoverflow
```

We won't perform this lab, but after the lab starts and you get the virtual terminal, take a look at the stack.c source code. This is a vulnerable program that the student is asked to exploit. Note the size of the `buffer` character array, which is derived from a random number generator seeded with your email address. The student must construct malicious input to compromise this program, and that input will depend on the size of the buffer. Post your value in chat so we can see the different values.

We will take a look at how this parameterization is achieved in a later section.

5 Deployment options (1:00-1:10)

Before taking a fifteen minute break, we'll review some options for deploying Labtainers for use in courses.

- The simplest deployment option is to have students download and install the VM appliance just as you have done.
- The VM appliance can be installed on a Virtual Desktop Infrastructure (VDI) solution such as VMWare Horizon. That provides students with access to the Labtainers VM desktop via their browsers, without having to install any software.
- The Docker Workstation product running on Macs or Windows computers can host Labtainers without installing the VM. Students then access a virtual desktop using their browser. Instructions for this are in the Student Guide.
- Labtainers can be deployed on cloud infrastructure, providing students with access to Labtainers via their browsers. While we do not provide a turn-key cloud solution, the Labtainer Instructor Guide has links to information on how to deploy Labtainers in the cloud, along with a sample cloud-config file for use with Canonical's Multipass cloud framework.
- Environments without Internet access can pre-load a VM with the desired labs, and then make that VM available to students within the isolated environment. The Instructor Guide identifies the script to use to pull the Docker images of the labs you wish to include.

6 Break (1:10-1:25)

7 Building new labs (1:25-2:10)

In this section of the workshop, we will walk through the steps necessary to create a new Labtainers lab. When creating new labs, you will often reference the Lab Designer Guide, so it would be good to have that document available. Your VM desktop should have a link to it. And it is available at `https://github.com/mfthomps/Labtainers/raw/master/docs/labdesigner/labdesigner.pdf`

Labs can be created either using a new `labedit` graphical user interface, or by using command line functions (or a combination of them). We will work with the GUI because it is easier and provides contextual help – and because its new and I want your help finding its bugs!

7.1 Start labedit

Enter the `labedit` command in any directory. You should see a GUI, and the first time you run the program, it will open the telnet lab. Before we go off and create a new lab, lets review some of the primary Labtainers elements as they are deployed in the telnetlab. If your UI does not have the telnetlab open, use `File=>Open lab` to open it.

Before we begin, use `File=>Preferences` to pick the text editor you would like to use when designing labs. The default is `vi` running in gnome-terminal:

```
gnome-terminal -- vi
```

If you'd prefer a GUI-type editor, change the field to `gedit`.

7.2 Review telnetlab implementation

The telnet lab includes two containers: a client and a server. And it includes one network. By clicking on a container, we can view and modify information about the computer, including its network connections. By default, the user password is the same as the user name, but that can be changed. Many of the options available by clicking the tabs will not apply to most labs.

Clicking a network displays the network's subnet and its external gateway. The containers use the external gateway to communicate with the host, and by extension the outside world. Individual containers can be configured to not permit external communication.

You can ignore the MAC VLANs.

The tap checkbox allows traffic on this network to be mirrored to a container designated as the network tap. This supports labs in which students will review network traffic traversing a network.

On the right hand side of the main UI are buttons for configuring automated assessment and for individualizing the lab.

The parameterization for the telnet lab is rather simple. This directs Labtainers to modify this file on the client container, replacing this string with a keyed hash. The parameter is assigned an identifier, which is referenced in the goals configuration, as we'll see in a bit.

Looking at the results configuration, we see flexible directives for identifying artifacts. This first one says to look in standard output from the telnet command and find the 4th token on the line starting with "my string is:". That artifact is assigned the *Result Tag* `fileview`. This next one is similar, but it looks at standard output from the ssh command. The last directive simply looks at this log file for a FAILED LOGIN string.

Goals can refer to Result Tags, and to parameter IDs as we see here. The first goal looks for a string match between the token value from a Result Tag, and the keyed hash from the parameter. The second goal is similar.

We'll look at more assessment and parameterization examples later. Now we'll dive into creating a new lab.

7.3 Create a lab

OK, I lied. Before we create a new lab, I'll note something you already know: Creating a computer science lab requires work. Labtainers does not drag-and-drop learning objectives. But if you have conjured up a lab, Labtainers helps you implement and deploy it. For our new lab, our learning objectives are basically those of the telnetlab. Only we'll make some improvements. And those of you who are particularly happy with their improvements (or at least see them as a step up from the admittedly old and somewhat neglected telnet lab) you can send me the lab and I'll consider updating what we distribute.

To create a new lab, use `File=>New Lab` and give it the name *mylab*. From the list of Base names, select the *wireshark2* base. We'll rename the container to *client* by right clicking on the container in the list. Though the lab is skeletal, it can be run.

Use `Run=>Build and run` to build and run the lab. Note that when we do this, the menu stays open, telling us that the framework is busy building the lab.

The mechanics of building a lab include creating Docker images of each container within the lab – in this case we have only one container. The build process then creates a running container using that image, and then performs run-time provisioning of the container, which is configured to appear as an independent computer.

When it is done, a terminal is created for interacting with computer that we created. We can confirm we have a Wireshark container by starting it:

```
wireshark &
```

When we built that lab, we did not see much happening. So lets view a couple of log files using the `View` menu. The labtainer.log displays a raft of debug statements, and errors when they occur. The docker_build.log reflects errors that Docker encounters when trying to build and run your containers. We plan to clean these logs up to eliminate much of the debugging noise. And we also plan to redirect important errors to the UI's output window. We can leave these log windows open in a corner for future reference.

Use `Run=>Stop lab` to stop the lab. We'll now add a server container, and select the *network.ssh* base. A summary of the different base images can be found in the Designer Guide.

We now have a client and a server computer. However they cannot yet talk because we have not created a network. Click the add button under Networks and provide a subnet, for example: `192.0.0.0/24`. Set an external gateway IP: `192.0.0.100`, and save it. Now that we have a network, we can assign it to our computers. Give each computer its own IP address.

We can now run our lab and confirm the computer have the assigned IP address and can be pinged.

7.3.1 Add software packages

Since we are doing a telnet lab and telnet is not part of our selected base images, we need to add the telnet client to our client and the telnet daemon to our server. Packages are typically added by modifying the container *Dockerfile*. This is a standard file used with Docker containers to define the file system image.

We can edit the Dockerfile of each container by right clicking on it and selecting `dockerfile` from the `Edit..` menu. For most labs, the only modifications you would make to the Dockerfile is add software packages. Looking into the file, we see a comment telling us to add our packages here. Since these are Ubuntu based containers, we use apt-get to manage packages. For the client, we add the telnet package.

For the server, the telnetd package.

7.3.2 Add files to a container

We will also want to add a couple of files to our server. This we will do manually at the command line. We open a shell by right clicking and selecting `Edit.. => Open shell on container`, which does as the name suggests. Note the Labtainers directory structure seen with `pwd`. This lab's directory has subdirectories for each Labtainers exercises. We are in a subdirectory for the server container.

Any file we create in this directory will appear on the computer when the lab is run. Lets create a file named `filetoview.txt` and add some text to it. We'll have the student cat the content of this file when performing the lab.

The second file we want to create is a system file needed by the xinetd daemon to properly start the telnet service when a computer connects to the telnet port. This telnet file needs to be in the `/etc/xinetd.d` directory. That directory will be created when the container starts executing – but that will be too late for us, so we'll create the directory beneath our `_system` directory which maps to the root directory in the running container.:

```
mkdir _system/etc/xinetd.d
```

The file itself is a standard Unix xinetd service file, which can be found on the internet – or cribbed from an existing lab. Since I know the telnet lab has such a file, I'll just copy that:

```
cp ../../telnetlab/server/_system/etc/xinetd.d/telnet _system/etc/xinetd.d/
```

Confirm the content is what we expect:

```
cat _system/etc/xinetd.d/telnet
```

We can now rebuild our lab and test our telnet connection. Note we don't need to stop the lab each time, the rebuild function will do that for us.

When the lab comes up, the server will be prepared to handle connections on the telnet port.

Confirm you can telnet from the client to the server. Then logout of the telnet session and start wireshark in the background.

```
wireshark &
```

Select eth0 as the capture device and then telnet again and watch your telnet packets in Wireshark. Enter `telnet` as the Wireshark filter to view only the telnet packets.

View the file that you put on the server, e.g., `cat filetoview.txt`. Then exit from the Telnet session and use ssh to login to the server:

```
ssh server
```

to confirm ssh access.

7.3.3 Modify /etc/hosts

To keep the lab simple and not distract the student with IP addresses, we'll update the `/etc/hosts` file so the server can be named with the word *server* instead of an IP address. Do this by clicking on the client container and then selecting the Hosts tab. We can either add explicit host/IP pairs, or direct it to add every host on a selected network, which is easiest.

7.4 Individualize the lab

Parameterization can be used to individualize labs for students so that two students are not likely to encounter the exact same lab. The motivation for this is to discourage sharing of solutions. Strategies for parameterizing labs are discussed in detail in our paper: *Individualizing Cybersecurity Lab Exercises with Labtainers* `https://nps.edu/documents/107523844/117289221/ComputingEdgeArticle.pdf` All Labtainers labs are individualized enough to prevent two students from simply submitting the same zip file. Our telnet lab is relatively simple and would not really benefit much from additional parameterization. But we'll add some to demonstrate the process.

We will modify the `filetoview.txt` file to contain a unique identifier for each student. And when we get to our section on automated assessment, we'll confirm the student displayed this unique string from both a telnet session and an ssh session.

7.4.1 Add a symbol to a lab file

From your server shell on the VM (not in a running lab), edit the `filetoview.txt` file to contain this line:

```
My unique string is: UNIQUE_STRING
```

7.4.2 Add a parameter directive

On the main labedit UI window, click the `Parameters` button to display the parameters for this lab. Give the parameter an identifier that we'll use later during assessment. We want to replace the symbol that we put into the `filetoview.txt` file with a hashed string, so selected `Hash Replace` as the operator type. Click in the file name field and select the server from the combo box and type `filetoview.txt` as the file name. Be sure to press enter so the text field is set. The symbol name we want to replace was `UNIQUE_STRING`, and this last field further seeds the keyed hash, it can be anything. Our parameter now reads as follows: Replace the symbol `UNIQUE_STRING` in the `filetoview.txt` file on the server container with a hash of a string that includes some stuff – and give the parameter a name of *unique_string* for future reference.

We can now rebuild our lab and confirm the symbol was replaced by a unique string.

7.5 Add assessment

For this lab, we'd like to know that the student viewed the `filetoview.txt` file from both telnet and ssh sessions. We'll start by identifying artifacts of interest. Click the `Results` button and then the `Add` button to create our first artifact identifier. Our first artifact will be evidence of viewing the `filetoview.txt` file while running telnet. Give the result an ID of `from_telnet`, which we'll refer to later from our goals definition. The artifact of interest will be on the client computer, because that is where the telnet program is run.

At this point we'll step back and review the naming of artifacts in Labtainers. Details of this naming are found in the `Lab Designer Guide`. Broadly, when a student runs selected programs, copies of standard input and standard output are captured within timestamped files. By default, we only do this for non-system programs such as programs the student writes or modifies. This narrow criteria avoids overwhelming the system with artifacts from common commands such as `ls`. However, there are times when we are interested in artifacts from system commands, such as our current lab where we'd like artifacts from telnet and ssh. We identify such commands by putting them into the `treataslocal` file for the container.

7.5.1 Treat as local

The obscure name of this file aside, it simply lists system commands whose artifacts are to be collected. Edit this file by right clicking on the container and selected `Edit => treataslocal`. Add telnet and ssh on separate lines and close the file.

7.5.2 Name the artifact file

Artifacts are named by combining the program name with either `stdin` or `stdout`. In our case, we are interested in standard output from the telnet command, so we enter `telnet.stdout` as the file name.

7.5.3 What from the file is of interest?

We want to find the unique string in this file so that we can compare it to what we expect for that student. So we'll treat each line as a set of tokens delimited by spaces. The line we entered into the `filetoview.txt` file had the unique string as the fifth word. And we identify the line as being the first line containing the string *My unique string*. Leave the timestamp type as `File`, that field is used for more advanced assessment.

We have now identified an artifact and given it the name `from_telnet`. We'll repeat the process to identify the ssh artifact, with the only difference being that we are interested in standard output from the ssh command.

7.5.4 Define goals

Click OK to save our results and then click the `Goals` button and add a goal. Our first goal is confirming the student displayed the file from telnet. Give the goal the name `did_telnet`. Labtainers promotes exploration, and as such we don't care when it is the student displayed the file from telnet, or if the student ran telnet a bunch of times without displaying the file. So we use a goal type of `matchany`, which will look for the artifact an any of the timestamped telnet.stdout files. We are looking for a string match of the artifact that named in `from_telnet`. And that artifact needs to match the parameter we defined in the `filetoview.txt` file. So set the `Answer type` to `Parameter`. That brings up our list of parameters, of which we only have the one.

This goals reads as: Look in all the timestamped `from_telnet` artifacts for a value that matches the parameter we generated for this student. If that is found, then the goal named `did_telnet` will be true, otherwise it will be false.

We now create a second goal for the ssh artifact. And save the goal definitions.

7.5.5 Test assessment

Now rerun the lab and perform the lab as a student is supposed to. As you do that, think about how tedious it can become to test a lab that includes many different steps – and hold that thought.

Once we've done the steps, we can use `Run => Checkwork` to check our assessment. The `checkwork` operation is available to students while they run the lab and it can be used either while the lab is still running or after it is stopped. If the `checkwork` results are not what you expected, one debugging strategy is to use the gradelab function as described later.

7.6 SimLab

The SimLab tool automates performance of lab exercises. We use it when developing and testing new labs. And we use it to perform regression testing on all labs as part of our continuous integration of the Labtainers product. And, as some of you who've been using Labtainers know, the SimLab scripts are the closet thing we have to instructor solution guides for many of the labs.

Use `Edit => SimLab directives` to open a shell in the simlab directory for this lab. By default, the tool looks for directives in a file named `simthis.txt`, so create that file here. Details of SimLab directives are in the `Lab Designer Guide`. We'll stick to a few simple directives. First, select the window to receive keystrokes, in our case the window title is `ubuntu@client`, so the directive is:

```
window ubuntu@client
```

Next, we want to telnet to the server, login and dump the file. We put those commands within `type_line` directives, and we add some `sleep` directives so we don't get ahead of the login exchange. Our resulting directives are listed below. Use of copy/paste on these commands may fail due to the PDF hiding new-line characters. Refer to the simthis.txt text file that accompanies this guide.

```
window ubuntu@client
type_line telnet server
sleep 2
# user name
type_line ubuntu
sleep 2
# password
type_line ubuntu
sleep 2
type_line cat filetoview.txt
type_line exit
sleep 2
#
```

```
# now ssh
#
type_line ssh server
sleep 2
# add to known hosts?
type_line yes
sleep 2
# password
type_line ubuntu
sleep 2
type_line cat filetoview.txt
type_line exit
```

Note all those `sleep` directives are only needed for interactive sessions where the program can get out of synch.

Once we have our SimLab directives defined and saved in the `simthis.txt` file, we can run SimLab. First make sure the lab is running, then use `Run => SimLab`. The tool uses a program called *xdotool* that simulates X11 keystrokes. It can be used for both command lines and GUI's (assuming the GUI has keyboard shortcuts.). When you run SimLab, take care to not click into any window on your VM, or the simulated keystrokes will appear on that window, with potentially disastrous consequences.

7.7 Assessment debugging with gradelab

When developing automated assessment for a lab, you will sometimes find that the assessment results do not match what you expect. Debug your assessment using the gradelab function. First stop the lab to ensure the artifacts are collected, then use the `. Run / gradelab` menu item to launch gradelab. Right click on the displayed URL to view the assessment results in a browser. Then select the table of goals.

Sometimes a good place to start debugging is to view intermediate results by clicking on the student name. That lets you see the different timestamped results. Click on the timestamp to view result tag values for that time. These are the tagged values defined in your results configuration, as found by the grader.

7.8 Temporal assessment

The automated assessment includes directives that let you specify temporal relationships between artifacts. For example, we could confirm that the student was running Wireshark during a telnet session. A more sophisticated example of temporal assessment can be found in the DMZ lab. There we want to confirm that the student ran the nmap command from two different computers with specific results, and did so between alterations of network filters. In other words, we might need to confirm the student was able to demonstrate several system properties without intervening changes to the system configuration.

8 Revisit lab examples (2:10-2:25)

Now that you are a bit more familiar with parameterization and automated assessment, we'll revisit a few of the labs that we looked at earlier. In your labedit UI, use `File / Open lab` to open the wireshark-intro lab.

8.1 wireshark-intro parameter and assessment

Click the Parameters button to view how this lab is individualized. The parameter simply replaces a symbol with a random value between zero and 99. This is done in the `fixlocal.sh` script, which runs on containers the first time they boot up. If we look at that fixlocal file, we see the random value controls how many frames to lop off the front of the PCAP that the student will view.

The assessment results have two result tags, one reflects the size of PCAP file the student saved, and the other reflects whether that PCAP includes the `john-password` string. Looking at the goals, we see an intermediate goal that is true if the size of the PCAP is less than 200 bytes. By convention, intermediate goals start with leading underscores, and they are not displayed as part of grading. That intermediate goal is then evaluated in a boolean expression along with the presence of the `john-password` string, and if both are true the *extracted_packet* goal is met.

8.2 bufoverflow parameters and assessment

Next open the bufoverflow lab. Looking at the parameters, focus on the first one. The other two simply define hashed strings the student will display. The first parameter controls the size of a buffer in a vulnerable program. We can open a shell in the container directory and look at the file being altered to replace the symbol: `BUFFER_SIZE`.

The first result tag reflects whether the student got a root shell from the vulnerable program and displayed the file readable only by root. The second tag is an example of the use of *precheck.sh* scripts. These scripts will run prior to each user command. When we look at the script, we see that it records the Address Space Randomization setting of the kernel. We'll not get too far into the weeds of the other results.

Looking at the goals, we see a boolean comparison that tells us if the student got root access while ASLR was on.

9 IModules (2:25-2:40)

After creating or customizing a lab, you can make that lab available to your students, and potentially to other instructors, by publishing the lab as an IModule. The *Lab Designer Guide* provides details on creating IModules.

Before you can publish new Labs, you need a Docker Hub registry, which you can create and manage at `https://hub.docker.com/`. You would assign that registry to your lab using `Edit / Config` and setting the registry field.

The general steps for creating an IModule include:

- Add your lab to a git repository
- Create a tar file of the run-time portions of your lab directory using the `create-imodules.sh` command.
- Publish that tar file on a web server accessible to your students.
- Push your lab's Docker images to Docker Hub using the Labtainers publish.py command.
- Your students then provide the url of your tar file to the Labtainers `imodule` command.

IModule creation is currently not supported in the GUI, so use the command line. We use git repositories to help manage the creation of IModule tar files. You'll notice we referred to *run-time* portions of the lab. That would exclude files that were used to create the lab, but are not necessary to perform the lab. For example, all the files that end up in a Docker image need not be included in the tar file. The `create-imodules.sh` tool manages that, assuming you've put your lab into a git repo. You need not be an expert with git to do that. It requires only a few commands as follows once your lab is mostly complete:

- run the `cleanlab4svn.sh` command to remove temporary files.
- cd to the labs directory, i.e., the parent of your lab directory.
- run `git initialize`
- run `git add mylab`

- run `git commit .  -m "my first lab"`

You then run the `create-imodules.sh` script to create the tar.

If you create an IModule that you think others might benefit from, send me the IModule URL and I'll publish it on the Labtainers web page and highlight it in the next Labtainers email.

10 Customized lab manuals (2:40-2:50)

In this section we will describe a way to provide your students with a custom version of a lab manual that they can reference from Labtainers. This does not require that you use the Labtainer VM or git. The example assumes you are customizing the telnetlab manual.

- Create your version of the manual in the pdf format (if the manual source is docx, export it as pdf). Many of our labs use Latex for the manuals, and thus editing those manuals will require a Latex system.
- Put that PDF manual in a file with the original name, in an otherwise empty directory structure that includes the name is the lab and "docs", for example:

```
cd ~
mkdir manuals
mkdir manuals/telnetlab
cd manuals/telnetlab
cp -aR \$LABTAINER_DIR/labs/telnetlab/docs .
telnetlab/docs/telnetlab.pdf
```

- In the above example, you would edit the docx file and then run make to create the PDF.
- Create a tar file of directory structures starting at the lab name. (Issue the tar file from the parent of the lab directory.)

```
cd ../..
tar cf imodule.tar telnetlab
```

- Publish that tar file onto a web server, i.e., something that responds to `http get` commands.
- Instruct your students to provide that URL to the `imodule` command.

If you wish to publish multiple custom lab manuals, put them all in the same tar file.

11 Conclusion and review (2:50-3:00)

We have covered a lot of material. Much more detail can be found in the *Lab Designer Guide*. Our goal was to give you an understanding of how Labtainers might benefit your courses, and how you can create and maintain lab exercises using the framework.

We'd like to spend the remaining time answering questions and hearing from you about the kinds of computer science lab exercises you think might fit well into the Labtainers framework.