2017

Labtainers: a Docker-based framework for cybersecurity labs

Irvine, Cynthia E.; Thompson, Michael F.; McCarrin, Michael; Khosalim, Jean

Irvine, C.E., Thompson, M.F., McCarrin, M., Khosalim, J., "Labtainers: a Docker-based framework for cybersecurity labs." 10`7 USENIX Workshop on Advances in Security Education, Vancouver, B.C., August 2017, 6 p.
http://hdl.handle.net/10945/56203

Labtainers: A Docker-based Framework for Cybersecurity Labs

Cynthia E. Irvine, Michael F. Thompson, Michael McCarrin, and Jean Khosalim
Department of Computer Science
Naval Postgraduate School
{irvine, mfthomps, mrmccarr, jkhosali}@nps.edu

Abstract

Successful lab designs are a valuable resource that should be re-used and shared among educators and between institutions. A collaborative, community-sourced design effort maximizes the benefit of the effort and expertise required to build and test an effective lab exercise. Unfortunately, infrastructure requirements, heterogeneous operating environments, and the desire to incentivize individual student work pose significant challenges that necessitate frequent updating, redesigning and retesting of assignments, creating a significant maintenance burden. To address these challenges, we present *Labtainers*: a container-based framework for the development, deployment and assessment of Linux-based cyber security lab exercises. Docker containers present a consistent environment that reduces the need for frequent updates, but with considerably less overhead than VM-based approaches. This enables a modest laptop to host labs consisting of multiple networked components. As such, the Labtainers framework is able to simulate a variety of security-relevant scenarios on a standalone student machine, without the need for elaborate infrastructure. Moreover, Labtainers' scripting support allows exercises to be customized on a per-student basis, then collected and evaluated automatically on the instructor machine. This capability enables the instructor to assign exercises where each solution is unique to the student with little or no increase in complexity of lab setup or assessment.

1 Introduction

Designing effective lab assignments requires the invention of exercises that are engaging and encourage interest, meet educational objectives of reenforcing material, are free from errors, and strike the correct balance between challenging students and discouraging them. A lab that succeeds in each of these areas takes a significant investment of time and expertise, and, more than likely, considerable testing through trial, error, and student feedback.

Educational materials for cybersecurity involve additional challenges. They are often sensitive to the details of particular implementations or operating environments. They may require infrastructure that can simulate interaction between multiple machines. These requirements impose significant maintenance costs: labs must be frequently updated, rebuilt and re-tested. Instructors must either develop hosting infrastructure, or handle the heterogeneity of student's personal machines. The former case involves considerable overhead in terms of labor and funding; in the latter case, it is difficult to simulate the wide variety of environments used by students to perform their labs.

Labtainers is a framework for developing and deploying Linux-based multi-component cybersecurity labs hosted entirely on a student's computer. We use Docker containers [3] to provide a controlled and consistent execution environment across all student computers regardless of the Linux distribution and configuration present on individual student platforms. 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 deployed in a single lab. Labtainers provide the advantages of a consistent execution environment without requiring one or more individual Virtual Machines (VM) per lab, and without requiring all labs to be adapted for a common Linux execution environment. A student laptop that struggles to run two or more VMs can readily run multiple containers simultaneously.

The framework includes automated assessment of student lab performance, and it supports individualizing labs to discourage sharing results between students.

As of this report, we have ported several labs from the SEED lab collection [6]. We have also developed some simple exercises to demonstrate features of the

framework. These include our multi-component `telnet` lab, which illustrates plaintext password transmission in `telnet` and mitigation of the vulnerability using `ssh`. We refer to this example lab throughout our overview of the framework in Section 2.

2 Labtainers Framework

In this section, we describe the Labtainer framework's support for individualizing labs and for automated assessment of student lab performance. Following, we cover networking support for multi-component labs.

2.1 Labtainer Actors

The Labtainer framework supports three main types of users, or *actors.*

The first is the Lab Designer, who is responsible for for creating the laboratory exercise so that it will meet its learning objectives. The lab designer determines if and how the lab is parameterized and whether automated assessment will be supported. The syntax used for lab configuration and parameterization can be found in the Labtainer Lab Designer User Guide. [16]

The second is the Instructor. This individual assigns the lab to the students and assesses their work. The instructor may or may not work with the lab designer to create the exercise.

Finally, the Student performs the laboratory exercises. Students are oblivious to the underlying framework that configures and individualizes their labs, and which will later gather any artifacts that may be required for assessment.

2.2 Using Labtainers

Students initiate a Labtainer exercise from any Linux system, (e.g., a VM on a laptop), that includes the Docker package. The *start* command names the lab, and the framework then pulls all necessary Docker containers from the Docker Hub and configures them for use by the student. Once configuration is complete, the student is presented with a set of virtual terminals that provide instructions and access to the lab environment. These terminals typically have `bash` shells via which the student interacts with the containers, which appear to the student as individual Linux systems, (e.g., clients, routers, servers), interconnected with one or more networks. The *stop* command collects a set of artifacts from the student's activity and places them in a zip file that the student then forwards to the instructor.

After gathering the zip files from all students for a given lab into a single directory, the instructor starts a special instructor container created for the lab. This container automatically assesses student artifacts and provides the instructor with a summary of each student's performance. The instructor is also provided with copies of each student's home directory and relevant artifacts along with an instance of the original lab execution environment, allowing inspection and review of student results.

Labs need not be designed for automated assessment or parameterization. But when they are, the lab designer performs this work, to the benefit of the instructor. Labs are designed primarily through use of configuration files, as illustrated in the discussion below. Several worked examples are available, as described in Section 4.

2.3 Individualizing labs for each student

Lab parameterization is intended to discourage students from sharing lab solutions, or finding solutions on the Internet. Labs are parameterized through symbolic substitution of values within the source code or data files that are part of the lab. The lab designer identifies these files, the symbols within the files, and the type of replacement that is to occur. For example, a symbol representing an array buffer size might be replaced by a random value bounded by a configurable minimum and maximum. Random values are created using a random number generator that is seeded with a string specific to each student and lab. (Seeds are created by concatenating a pre-defined string for the lab with the student's email address.) In the case of a buffer size parameter, the seed would be used to set the buffer size in that student's instance of the lab.

Parameterization can be used to change computations. For example, changing the buffer size might affect how a buffer overflow is crafted. It can also be used to customize stored artifacts. In our `telnet` lab, the student is directed to `telnet` to a server and display the content of a specific file. Parameterization causes the content of this file to be unique to the student, e.g., containing the results of a hash keyed with the student's unique seed. For example:

```
FSTRING : HASH_REPLACE : \
  telnetlab.server.student=filetoview.txt : \
  TELNET_STRING : mytelnetfilestring
```

causes the symbol `TELNET_STRING` in the file `filetoview.txt` to be replaced with a hash of the string "mytelnetfilestring" keyed with the student's lab-specific seed.

As will be seen below, the assessment configuration file syntax includes an ability to name parameter symbols such that the assessment function automatically compares the results from each student's artifacts with values generated specifically for that student.

2.4 Automated Assessment of Goals

Labtainer automated assessment functions provide instructors with binary indicators of student achievement of specific goals. While future work may incorporate the ability to define higher-level evaluation logic, the feedback currently provided is somewhat similar to forensic indications of specific activity. As such, the goals tracked by automated assessment simply reflect whether specific inputs or outputs were generated. This functionality is not intended to fully assess student comprehension or performance; rather, it serves as an aid to instructor judgment. Depending on the lab and the instructor, metrics generated by automated assessment may be sufficient to grade a lab. Alternatively, they may be viewed as broad indicators of progress, or confirmation that the student engaged with the lab environment in addition to writing a lab report. In addition to measurement of individual student progress, suitably designed Labtainer goals aggregated across students might also highlight difficult or problematic areas of the lab. These could be used to identify areas in which the assignment or instructional materials could be improved in the future.

The assessment functions do not track time spent performing the lab. This is deliberate, because our intent is to promote exploration by students at their own pace. To further encourage exploration, the typical manner of Labtainer goal assessment will indicate that a given goal has been met so long as there is at least one indication of its having been met, regardless of the quantity of failures that precede or follow that event.

Goals are defined in terms of artifacts gathered from the student lab sessions. These artifacts include the entire home directory of the student containers, selected system files identified by the lab designer, and files containing the `stdin` and `stdout` streams from student interactions with selected programs. The framework captures copies of `stdin` and `stdout` by using the student's `.profile` to hook the `bash` shell with functions similar to the ZSH *preexec* and *precmd* functions [7] . These functions intercept all `bash` commands, allowing augmentation of commands before they are executed. The framework causes selected commands to use the `bash` *tee* function to make copies of `stdin` and `stdout` into timestamped files.

The lab designer locates results within the artifacts by assigning their symbolic names in the configuration file settings. For example:

```
fileview = client:telnet.stdout : \
  4 : STARTSWITH : My string is:
```

assigns the symbol `fileview` a value equal to the fourth space-delimited token on the first line that starts with "My string is:" within `stdout` of the program named `telnet`. Since there may be many instances of `stdout` files from invocations of `telnet`, the framework maintains a set of `fileview` symbols, one per timestamp.

The results extracted from student artifacts are compared to expected values to determine whether goals have been met. Goals evaluate as *true* or *false*, and are defined in configuration file entries. For example, the entry

```
telnetview = matchany : \
  string_equal : fileview : \
  parameter.FSTRING
```

will indicate that the student achieved the `telnetview` goal if any of the timestamped `fileview` symbols match the value of the FSTRING parameter. In this example, the FSTRING parameter is unique to each student, as described in the previous section.

2.5 Networks of Containers

This section describes use of Labtainers to create a simple network of containers consisting of client and server computers.

Each container within the lab is defined by a Dockerfile which specifies the packages and files within the file system of the container image. The Labtainer baseline image includes a set of packages useful for many labs, including common development tools such as `gcc`, `vim` and `python`. All lab-specific Dockerfiles reference this baseline image, or an image derived from that baseline. The Dockerfile then identifies additional packages and files for the container. In this example, the client container includes the `telnet` package. The server container baseline image includes the `xinetd`, `sshd` and `rsyslog` services. The server's lab-specific Dockerfile builds on this image to also include the `telnet` service. Outside of Labtainers, typical Docker containers do not include multiple services, and their logs are forwarded to the host and collated with other container logs. Moreover, a Dockerfile typically starts a single service using the `ENTRY` directive. Labtainers are not conformant with this model because our goal is for the containers to appear as typical Linux systems. The `ENTRY` directive for our example server container starts a simple script that launches `rsyslog` and `xinetd`. The former causes system log entries to appear in their familiar locations within `/var/log`, and the latter launches the `telnet` and `sshd` services in response to incoming network connections.

Docker images generated from Dockerfiles for each of the lab's containers are implicitly referenced in the *start.config* file created by the lab designer for each lab. This file identifies the containers and defines the networks within the lab. The configuration file entry for our example network is:

```
NETWORK SOME_NETWORK
      MASK 172.20.0.0/24
```

```
        GATEWAY 172.20.0.100
```

A container connects to networks by naming the networks in the configuration file entry for that container. For example:

```
CONTAINER client
        USER ubuntu
        TERMINALS 2
        SOME_NETWORK 172.20.0.2

CONTAINER server
        USER ubuntu
        TERMINALS 1
        SOME_NETWORK 172.20.0.3
```

The container names of client and server resolve to their corresponding Dockerfiles per the Labtainer naming convention. These entries assign network addresses to the containers, and define the number of virtual terminals to be created and attached to each container when the lab runs.

These three configuration file entries suffice to define the simple network seen by students when performing the example lab. When the lab starts, the virtual terminals are created and present `bash` shells, allowing the student to interact with the containers which appear to be independent Linux systems connected by a network. The server container offers the `telnet` service, which the student can reach by issuing a `telnet` command from the client `bash` shell. All of the students will see the very same `telnet` server and client, regardless of the Linux distribution they are running, and regardless of what packages are installed on their Linux hosts.

Since the purpose of this example lab is to highlight the fact that `telnet` passes passwords over networks in clear-text, the `tcpdump` utility is available on the server container for use by the student to observe network traffic. When the student starts the `tcpdump` program, its `stdout` is automatically captured within timestamped files as described in Section 2.4. If the student is directed to attempt a `telnet` login with a specific password, e.g. `plaintextpassword`, that password will appear in the `stdout` file. As described in the Section 2.4, the designer could define a goal corresponding to the presence of the prescribed password in a `tcpdump stdout` artifact. Though quite simple, such a goal would indicate that the student started the `tcpdump` program on the server, and then attempted a `telnet` login. This limited, though potentially informative, automated assessment of the example lab is realized through two configuration file entries. This entry in the *results.config* file:

```
password_on_wire = tcpdump.stdout : \
  CONTAINS : plaintextpassword
```

and this entry in the *goals.config* file indicates not only that the student ran `tcpdump`, but that `plaintextpassword` was on the wire:

```
ran_tcp_dump = is_true: password_on_wire
```

3 Discussion

In this section, we provide a brief overview of related work, contrast various approaches to providing laboratory exercises, and discuss the limitations of Labtainers.

3.1 Related Work

Time and infrastructure resource requirements often compel security instructors to seek lab support from centralized security lab projects such as DeterLab [11]. RAVE (Remote Access Virtual Environment) [12, 20], and EDURange [19]. We note that the Tele-Lab project [21] is similar to these, but offers only test accounts. In contrast to all of these, which require students to connect to the infrastructure platform, Labtainers frees students to work unconnected, thus further encouraging self-paced and intermittent activity. In addition, containers afford more fine-tuned lab environments and are simpler for instructors to manage and deploy.

The SEED project [5, 4, 6] has developed 33 freely available labs in three categories: vulnerability and attack, design and implementation, and exploration. Complementing these, Wang has developed a set of lab exercises for IT security [18]. These labs are not parameterized, neither do they support automated assessment.

Parameterization of security labs was incorporated into Tele-Lab. [21] In contrast to Tele-Lab, where parameters are predefined and stored in a parameter database, Labtainers parameterizes each lab by using metadata associated with the student. PolyLab randomizes lab exercises by using hashes. [8], but this framework does not support the virtualization provided by Labtainers.

Ala-Mutka surveyed automated assessment technologies used in programming courses [1], e.g. [10, 14, 13]; however, none were directly applicable to Labtainers.

3.2 Why not VMs?

Several alternatives are available to instructors who wish to offer cybersecurity labs: hands on experience involving physical machines, virtual machines hosted on an infrastructure-as-a-service (Iaas) platform, virtual machines hosted on each student's laptop, and containers executing either on the student's Linux host or in a Linux virtual machine hosted on the student's system.

Hay et al. suggested the use of virtual machines to support security labs [9]. The advantages of containers were discussed in Section 3.1. Some virtual machine challenges solved by using containers are discussed below.

On demand cloud computing resources, such as Amazon Web Services (AWS) [2] require special permission to run many simple network security exercises, such as

port scans and penetration testing. [1]

Construction of an institutionally-owned and operated virtual machine farm is likely to require considerable initial hardware investment and technical expertise, as well as an ongoing operational tail for maintenance, user management, continuity of operations, and backups. A proprietary system for managing VMs, such as vSphere [17], usually requires local expertise and a support agreement, while less costly open source options, such as KVM [15], require even greater levels of institutional expertise. If students are required to host a number of VM images on their personal computers or laptops, the resource requirements can quickly exceed what is available on the host. In contrast, Linux containers [22] offer a less costly and less complex alternative that affords lab designers and instructors greater control, without not tethering students to a server farm.

Where the physical component cannot be virtualized, the solution may involve some combination of approaches, both physical and virtual networked together. For example, the container could be connected through inter-virtual machine networking to a virtual machine running Windows.

3.3 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 as described in Section 2.5. However, other process trees will appear as they do in a 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 the Labtainer scripts that manage capture and collection of artifacts. In addition, that directory contains the `stdin` and `stdout` files generated by student actions. Further, 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.

Our future work includes porting more labs, whether on bare machines or in virtualized environments, to Labtainers. In so doing, we will explore the limitations of Dockers support for various security labs. For example, we believe that a lab on *iptables* is possible, but we do not know what is impossible other than heterogeneity for the underling kernel. Another area of future work is construction of a grammar for the lab specification language.

4 Availability

Our initial release of the Labtainers framework includes worked examples for several labs, many of which were derived from SEED labs [6, 4]. These include:

- **Format String** Derived from the SEED *Format String Vulnerability Lab*, this lab gives students first hand experience exploiting vulnerabilities associated with the *printf* function. The lab is parameterized such that one of the "secret" values displayed by the exploited program is a random displayable ascii character. Automated assessment confirms the student performed each of the tasks identified in the original SEED lab.

- **Buffer Overflow** Derived from the SEED *Buffer Overflow Vulnerability Lab*, this lab requires the student to craft a data file that exploits a buffer overflow when consumed by a vulnerable program. The lab is individualized by changing the size of the buffer to be overflowed, and by changing the content of a file the student is asked to display after gaining a root shell. Automated assessment confirms the student displayed the target file while executing the vulnerable program. And it confirms the student took actions consistent with exploring stack guards, as directed by the original SEED lab.

- **One Way Hash** Derived from the SEED *One-Way Hash Function and MAC*, this lab introduces the student to hash functions offered by the openssl program. It highlights a simple use of parameterization and goals to confirm the student turned in his or her own zip file without individualizing any other aspects of the lab. It also demonstrates the ability to enumerate several goals and then use a counting operation to confirm that the student generated hashes within at least N of the available hash algorithms as prescribed by the SEED lab.

[1] `https://aws.amazon.com/premiumsupport/knowledge-center/penetration-testing/`

- **Telnet** An implementation of the simple example described in Section 2. It demonstrates a lab with multiple networked containers.

- **Openvpn** The student configures the *openvpn* application to create an encrypted tunnel between a client and a server, through a router. The student then runs tcpdump to observe encrypted and unencrypted traffic. This lab illustrates the use of a simple router implemented within a container.

The Labtainer framework and user guides are available at:

`http://my.nps.edu/web/cisr/labtainers`

Acknowledgements

This work was supported by NSF grant DUE-1140938. The views expressed in this material are those of the authors and do not reflect the official policy or position of the National Science Foundation, Department of Defense or the U.S. Government.

References

[1] ALA-MUTKA, K. M. A survey of automated assessment approaches for programming assignments. *Computer Science Education 15*, 2 (June 2005), 83–102.

[2] AMAZON.COM. About aws. https://aws.amazon.com/about-aws/ Retrieved 8 May 2017, September 2011.

[3] ARVAM, A. Docker: Automated and consistent software deployments. https://www.infoq.com/news/2013/03/Docker, 27 March 2013.

[4] DU, W. Seed: Hands-on lab exercises for computer security education. *IEEE Security and Privacy Magazine 9*, 5 (Sept. 2011), 70–73.

[5] DU, W., JAYARAMAN, K., AND GAUBATZ, N. B. Enhancing security education with hands-on laboratory exercises. In *Proceedings 5th Annual Sympoisum on Inforamtion Assurance (ASIA'10)* (June 2010).

[6] DU, W., AND WANG, R. Seed: A suite of instructional laboratories for computer security education. *J. Educ. Resour. Comput. 8*, 1 (Mar. 2008), 3:1–3:24.

[7] FALSTAD, P. An introduction to the Z shell. http://zsh.sourceforge.net/Intro/intro_toc.html Last accessed 9 May 2017, 30 November 1995.

[8] GIACOBE, N. A., AND KOHLER, R. Development of polymorphic homework and laboratory assignments in cyber security with PolyLab. In *NICE (National Initiative for Cyber Education) Conference 2016* (Kansas City, MO, November 2016).

[9] HAY, B., DODGE, R., AND NANCE, K. Using virtualization to create and deploy computer security lab exercises. In *Proceedings of The IFIP Tc 11 23rd International Information Security Conference: IFIP 20th World Computer Congress, IFIP SEC'08, September 7-10, 2008, Milano, Italy*, S. Jajodia, P. Samarati, and S. Cimato, Eds. Springer US, Boston, MA, 2008, pp. 621–635.

[10] IHANTOLA, P., AHONIEMI, T., KARAVIRTA, V., AND SEPPÄLÄ, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2010), Koli Calling '10, ACM, pp. 86–93.

[11] MIRKOVIC, J., AND BENZEL, T. Teaching cybersecurity with DeterLab. *IEEE Security and Privacy 10*, 1 (Jan. 2012), 73–76.

[12] NANCE, K., TAYLOR, B., DODGE, R., AND HAY, B. Creating shareable security modules. In *7th World Conference on Information Security Education* (9-10 June 2011), R. C. Dodge and L. Futcher, Eds., vol. 406 of *IFIP Advances in Information and Communication Technology*, pp. 156–163.

[13] PETTIT, R. S., HOMER, J. D., HOLCOMB, K. M., SIMONE, N., AND MENGEL, S. A. Are automated assessment tools helpful in programming courses? In *122nd ASEE Annual Conference & Exposition* (2015), American Society for Engineering Education.

[14] PIETERSE, V. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research* (Open Univ., Heerlen, The Netherlands, The Netherlands, 2013), CSERC '13, Open Universiteit, Heerlen, pp. 4:45–4:56.

[15] SHAH, A. Ten years of kvm. https://lwn.net/Articles/705160/, 02 November 2016.

[16] THOMPSON, M. F. Labtainer lab designer user guide. `http://my.nps.edu/documents/107523844/109121513/labdesigner.pdf`, 23 May 2017.

[17] VMWARE. vsphere and vsphere with operations management. http://www.vmware.com/products/vsphere.html, April 2017.

[18] WANG, X., BAI, Y., AND HEMBROFF, G. C. Hands-on exercises for it security education. In *Proceedings of the 16th Annual Conference on Information Technology Education* (New York, NY, USA, 2015), SIGITE '15, ACM, pp. 161–166.

[19] WEISS, R., MACHE, J., AND LOCASTO, M. Edurange: Hands-on cybersecurity exercises in the cloud. *J. Comput. Sci. Coll. 30*, 1 (Oct. 2014), 178–180.

[20] WEISS, R., NESTLER, V., LOCASTO, M. E., MACHE, J., AND HAY, B. Hands-on cybersecurity exercises and the RAVE virtual environment (abstract only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, pp. 759–759.

[21] WILLEMS, C., AND MEINEL, C. Online assessment for hands-on cyber security training in a virtual lab. In *Global Engineering Education Conference (EDUCON)* (apr 2012), IEEE.

[22] YU, Y. *OS-level Virtualization and its Applications*. PhD thesis, State University of New York, Stony Brook, Stony Brook, NY, December 2007.