NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A PARAMETERIZATION FRAMEWORK FOR CYBERSECURITY LABORATORIES

by

Jean Khosalim

March 2017

Thesis Advisor: Cynthia E. Irvine
Co-Advisor: Michael F. Thompson

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
IMPLEMENTATION OF A PARAMETERIZATION FRAMEWORK FOR CYBERSECURITY LABORATORIES

5. FUNDING NUMBERS

6. AUTHOR(S) Jean Khosalim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Computer Science courses often include laboratory exercises to make sure certain concepts are experienced hands-on by the students. These courses sometimes are taken by a large number of students and each assignment needs to be graded. Instructors or teaching assistants responsible for grading assignments are presented with the tedious task of verifying students' work. Besides making sure that each student performs the assignment correctly, the assignment grader may also be concerned that students do not cheat on the assignment by copying and submitting work from other students.

The objective of this thesis is to investigate and develop a framework for Linux-based cybersecurity laboratory exercises performed on individual student computers. The purpose of the framework is to provide the designer of laboratory exercises with tools to parameterize labs for each student, and automate some aspects of the grading of laboratory exercises. A prototype of this framework was implemented by making use of the Linux Containers, which provide an additional benefit of standardizing execution environments utilized by students and instructors.

14. SUBJECT TERMS automated assessment tool (AAT), parameterization framework			**15. NUMBER OF PAGES** 77
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

IMPLEMENTATION OF A PARAMETERIZATION FRAMEWORK FOR CYBERSECURITY LABORATORIES

Jean Khosalim
Civilian, Department of the Navy
B.S., University of California, Los Angeles, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2017

Approved by: Cynthia E. Irvine
Thesis Advisor

Michael F. Thompson
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Computer Science courses often include laboratory exercises to make sure certain concepts are experienced hands-on by the students. These courses sometimes are taken by a large number of students and each assignment needs to be graded. Instructors or teaching assistants responsible for grading assignments are presented with the tedious task of verifying students' work. Besides making sure that each student performs the assignment correctly, the assignment grader may also be concerned that students do not cheat on the assignment by copying and submitting work from other students.

The objective of this thesis is to investigate and develop a framework for Linux-based cybersecurity laboratory exercises performed on individual student computers. The purpose of the framework is to provide the designer of laboratory exercises with tools to parameterize labs for each student, and automate some aspects of the grading of laboratory exercises. A prototype of this framework was implemented by making use of the Linux Containers, which provide an additional benefit of standardizing execution environments utilized by students and instructors.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AAT	automated assessment tool
APAC	automatic programming assignment checker
ASLR	address space layout randomization
CBA	computer-based assessment
GUI	graphical user interface
RBAC	role-based access control
VM	virtual machine

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to both my advisors, Dr. Cynthia Irvine and Mike Thompson, for providing their invaluable time and guidance throughout this entire project. I am grateful to Mike for his patience and technical expertise during the prototype implementation.

This material is based upon work supported by the National Science Foundation under Grant No. 1438893.

Finally, I would like to share the credit for the successful completion of this thesis with my wife and my daughter, who both were a source of constant support and encouragement.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Laboratory exercises are an integral part of teaching computer science concepts to students, by providing them hands-on experience. These kinds of activities also allow instructors to evaluate how well students understand concepts. As computer science courses have grown in popularity at universities and colleges, so has the burden on instructors to grade laboratory exercises, which can be tedious and time-consuming. Instructors and teaching assistants who are responsible for assessing laboratory assignments face three inherent burdens: 1) ensuring students submit original work, 2) verifying that the work submitted is accurate, and 3) distinguishing between failures to comprehend concepts and failures related to computer administration and provisioning.

B. PURPOSE OF STUDY

The purpose of this thesis is to investigate strategies to lessen the three burdens identified above, and implement the results within a framework for computer science laboratory exercises that focuses on cybersecurity.

The following question will be examined in this thesis: What kinds of automated support might assist the designer of laboratory exercises to achieve the following?

1. Verify that students performed lab exercises, with some identification of areas, specifically, to easily determine if there are the portions of a lab that many students struggle with.
2. Gain confidence that the students did their own work, and did not obtain their exercise results from other students or the Internet.
3. Provide all students with an identical environment in which to conduct the lab exercise to help ensure that student failures and frustrations are not due to administrative and configuration problems.

C. ROLES

For the purpose of this thesis, there are three different roles that would interact with the framework.

1. Student Role

The student role is any person performing the laboratory exercise. This is the intended audience that should benefit or gain additional knowledge related to computer science by actually performing the tasks described in the laboratory exercise. The framework targets cybersecurity lab exercises that occur in a Linux environment, and the student should experience a typical Linux environment that is not visibly altered by the implementation of the framework.

2. Instructor Role

The instructor role is the person who provides guidance to the students. Any prior knowledge that is required to perform a laboratory exercise by the students is the instructor's responsibility; i.e., the instructor must provide enough guidance for students to complete the laboratory exercise successfully. The framework features are not intended to provide instruction, rather they provide an environment for exploration and experimentation related to cybersecurity concepts.

3. Laboratory Designer Role

The laboratory designer role is the person who actually creates the laboratory exercise. The laboratory designer might also be an instructor. The laboratory designer may collaborate with instructors as part of creating a specific laboratory exercise so that the exercise covers specific computer science concepts.

D. HIGH-LEVEL GOALS

There are several high-level goals that the parameterization framework intends to achieve; they are given in the following sections.

1. Parameterization Support

A concern among instructors is that some students will cheat, if the opportunity is available and easily obtainable. One simple example is when exercise results are the same for all students. The first student who has finished an exercise can then easily pass the results to other students. One of the goals of the framework is to allow an exercise to be

tailored such that results are different for each student performing the exercise. The goal is to provide a simple barrier for cheating. The framework is will not attempt to address sophisticated attempts to cheat (e.g., ones that subvert the framework itself).

An example of this *parameterization* is an exercise to exploit a buffer overflow in an existing program. If the framework can cause the buffer overflow threshold to differ between students, then simple cheating by many can be prevented.

2. Default Automatic Assessment

The need for grading lab exercises can be burdensome for instructors. In addition, the common strategy of requiring students to answer questions in essay form does not always give instructors confidence that students actually performed the steps of the lab exercise. The framework can (and should) provide instructors with a full copy of each student's lab environment to review. However, another goal of the framework is to promote student exploration while they are performing computer security laboratory exercises. The results of such exploration can complicate the instructor's view of the student's work due to additional files, some of which may have been created by the student for purposes tangential to the point of the lab. The framework should provide support for lab instructors to automatically assess some aspects of a student's lab activity.

The framework should allow lab designers to express lab exercises as steps that students are to perform. Automated grading should then provide the instructor with evidence that the student performed each step of an exercise to help instructors assess the level of learning by each student. The framework should automate collection of artifacts for review by instructors.

3. Consistency

Differences in execution environments can frustrate students and instructors. An example might be student whose Linux environment includes a library that is different from the version used by the instructor and other students, resulting in that student's failure to complete a lab exercise. Use of an identical laboratory exercise environment for both students and the instructor can eliminate potentially time-consuming provisioning

tasks, and will allow students to spend more time on the lab exercise and less time on system administration tasks.

The framework should also allow the instructors to review students' work in an execution environment similar to the students' environment. This would help the instructor repeat steps that the students performed for a specific part of the laboratory exercise (e.g., to review problems the student might be encountering).

4. Lab Environment

The framework is intended to support cybersecurity laboratory exercises conducted in Linux environments. The framework assumes students will work on their own computers, or on individual lab computers. The framework should not rely on centralized servers, such as VM farms, to host the lab exercises. Individual student computers of modest capabilities should be able to host labs built using this framework.

E. ORGANIZATION OF THESIS

This thesis is organized as follows: Chapter I addresses challenges related to computer science laboratory exercises, and provides the motivation and purpose of this thesis. Chapter II presents background information related to automating grading of lab exercises and strategies for obtaining consistent execution environments. Chapter III provides the project description, which includes the concept of operation and the methodology for designing and implementing the parameterization framework, and it describes how the concept of operations affects requirements for the prototype implementation. Chapter IV covers the implementation of the parameterization framework. Chapter V concludes the thesis with suggestions for future work. Sample use of the parameterization framework using actual laboratory exercises is provided in Appendix C.

II. BACKGROUND

This chapter contains background information on topics relevant to the implementation of the parameterization framework. The chapter begins with a review of existing automated assessment tools, to understand what others have developed and determine if those tools could contribute to the framework. The chapter also discusses SEED labs [1] and how the sample labs are relevant examples for the prototype framework. This chapter also provides an overview of Linux containers [2], specifically Docker [3].

A. AUTOMATED ASSESSMENT TOOLS

There have been many automated assessment tools (AATs) created to help assess how students perform in programming courses. In the paper, "Are Automated Assessment Tools Helpful in Programming Courses" [4], there were 63 automated assessment tools or comparable tools listed. From that list of tools, a number of tools were investigated and studied. These tools were selected because they appear to perform automated assessment of student lab work. Because all of these tools are intended to assess programming exercises, none of them provides the basis for our framework. However, we investigated the tools to see if they employ techniques that would benefit the framework implementation.

These AATs incorporate combinations of two techniques discussed by Ala-Mutka [5] for automated assessment of programming exercises:

- Dynamic Assessment techniques where the assessment or grading is based on executing the student code against test data.
- Static Assessment techniques where the student's code is analyzed programmatically, such as coding style, logic or design, and the assessment or grading is based on the information obtained from that static analysis.

1. Autograder

Autograder [6] automatically grades students' programming labs. The Autograder is a dynamic assessment technique. A set of test input is provided to the student's

program and the resulting output from the program is checked against a baseline or expected output. An example laboratory exercise is for the student to write a program to compute the distance between two points. The program is then given a set of inputs in the form of the X and Y coordinates of the two points, and the program should output the distances between the two points in the units of that coordinate system.

None of the features found in Autograder seem to apply to the parameterization framework. However, individual lab designs may incorporate dynamic assessment techniques, if desired.

2. CourseMarker

CourseMarker [7] is the successor of Ceilidh [8]. Both were developed by the University of Nottingham as Computer Based Assessment (CBA) systems. Students and instructors interact with CourseMarker through a web interface. The CourseMarker system provides grading tools that support course work exercises, ranging from programming exercises to multiple-choice exercises. Ultimately, the grading process depends on accurate specification of the questions for the exercises. For example, the student's program or solution is assessed based on the accurate specification of the format of the input to the student's program and the resulting output created by the student's program given the specific input. For programming courses, at the discretion of the teacher, a skeleton solution may be provided to the student. Once the student has developed a solution, it can be submitted for assessment. Marking tools for assessment of programming exercises range from typographic layout to program complexity. The CourseMarker system is quite complicated, as shown in Figure 1.

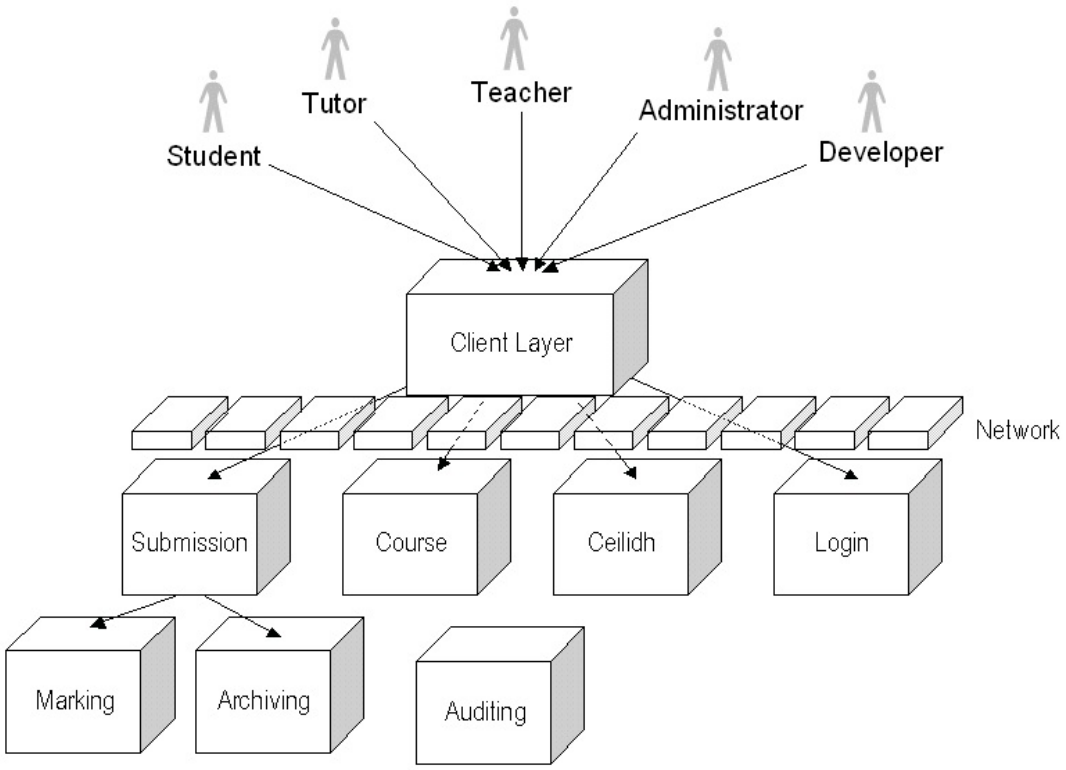

Figure 1. A High-Level View of CourseMarker. Source: [7].

In CourseMarker, its parameterization constrains the exercise's properties in various ways, such as the maximum allowed number of submissions, the maximum CPU time allocated to run the student's solution, or other resources used for assessment. Each student gets the same configuration for the exercise; that is, the exercise itself is not parameterized. To detect plagiarism, CourseMarker compares all students' solutions to determine if there is evidence of plagiarism based on similarities of the students' work.

There are no features in CourseMarker applicable to our parameterization framework.

3. Automatic Programming Assignment Checker

Automatic Programming Assignment Checker (APAC) [9] is very similar to Autograder in terms of its functionality. APAC makes use of Linux containers, specifically Docker. Students perform the laboratory exercise in their own environments.

When a student finishes an exercise, the result is the student's program, which is submitted to the APAC system through the web interface. A container is spawned to run the student's program to generate the output to be compared with the expected or reference output. The output similarity is computed and then multiplied with the maximum point allowed to generate the grade for each student for the exercise.

Although APAC is written in Java, it can handle exercises where the students' programs are written in C, C++, C#, Python, or Java. Similar to Autograder, APAC will run the students' program given a set of inputs, and the assessment or grading is based on comparing the output similarity to the expected output by computing the Levenshtein distance (i.e., a measure of similarity between source string S and target string T, with the distance as the number of deletions, insertions, or substitutions required to transform string S to string T).

While this tool uses Docker containers, it is service-oriented and not applicable to our parameterization framework.

4. GROK Learning System

The GROK Learning system [10] is a commercial product. Students access the GROK learning system through a web interface.

To use the GROK Learning system, students only need a web browser. The students do not perform the exercises in their own environment. Instructors specify the description of the exercise and the students write their code using their browsers. The GROK system provides feedback to the student such as compiler or syntactic errors. By testing the freely available sample, we concluded the functionality of the GROK learning system is very similar to the Autograder, in the sense that the students will be asked to program a set of exercises and students will test their programs with a given set of inputs to see if the expected output is achieved. When the students submit their programs, different sets of inputs and expected outputs (most likely corner test cases) are tested against the programs. Trial submissions are allowed. Test cases pass if the expected output is returned by the program given a specific set of inputs. Grading is based on how many test cases pass or fail.

The ability for students to submit trials may be a useful feature for our parameterization framework.

5. ASSYST

The ASSYST system [11] is intended to grade students' programs based upon five metrics: correctness, efficiency, style, complexity, and test data adequacy, as shown in Figure 2.

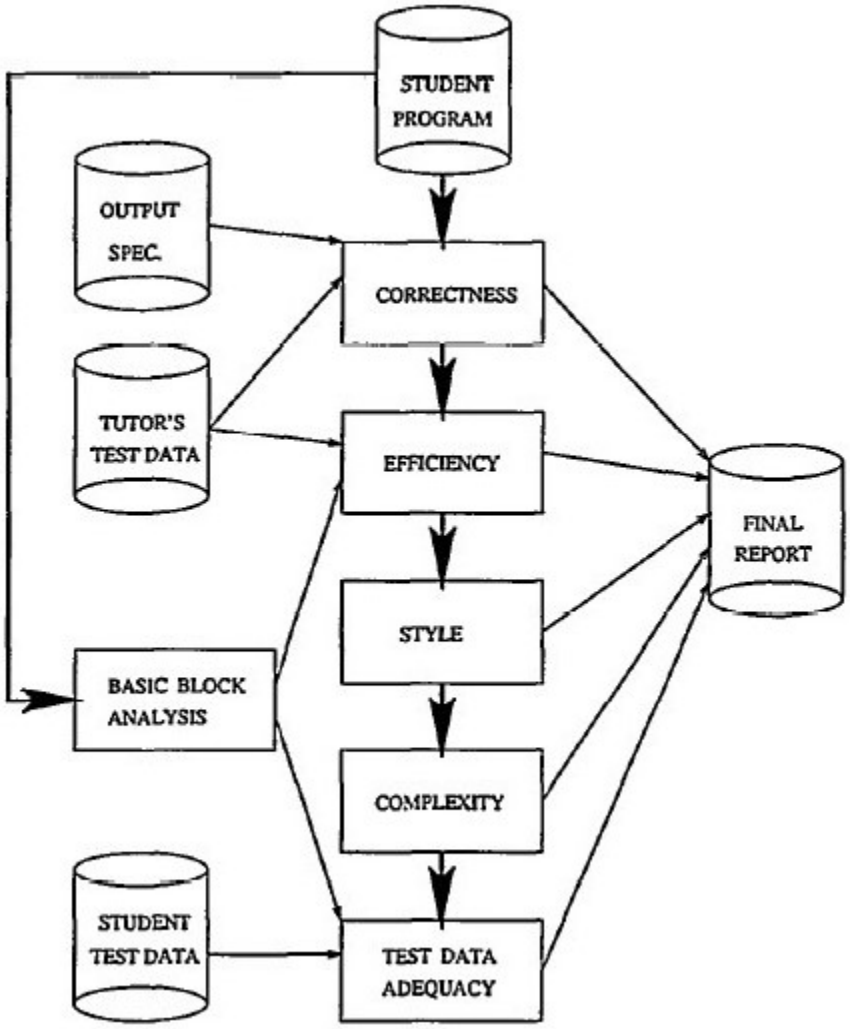

Figure 2. ASSYST Assessment Process. Source: [11]

ASSYST is considered a hybrid system, where part of the marking process is automated and part is performed manually by the instructor. The correctness metrics are obtained by performing pattern matching; in essence, the output of the student's program, given the input of the instructor's test data, is compared to the output specification from the instructor. For efficiency metrics, the student's program is fed the instructor's test data and its efficiency is measured by the CPU time to run the student's program. The ASSYST system also performs a static analysis of the program code (structure in terms of basic code blocks, count of the number of statements in each block, etc.). The style metrics for C programs are based on the program characteristics, such as module length, number of comment lines, and use of indentation. The instructor may alter the final value for the style metrics. For complexity metrics, McCabe's metric is used. Students submit their programs along with their test data. The test data are evaluated based on the test data's coverage. The result of all the metrics above is used to construct the final report for each student's grade.

The ability for the instructor to manually inspect student work may be useful for our parameterization framework.

6. VMChecker

The VMChecker [12] provides a web-based graphical user interface (GUI) for administration and grading (i.e., it provides the capability to accept students' submissions, but the grading is manually performed by updating or entering grade information for each student). The automated checker portion is related to functionalities such as applying penalty points if the submission is after the deadline.

Incorporating late penalties may be interesting, but our framework does not address either the mechanics of student submission or the grading policies.

7. Web-CAT

The Web-CAT [13] is a web-based system that allows "Automated Grading Using Student-written Tests." Web-CAT is used for exercises where students write tests on their own to measure correctness of their solution and their test cases are tested on

other students. A sample assignment provided to a guest user is for the user to implement a Java Calculator. Students will log in through the web interface and submit the assignment solution along with test cases (Web-CAT supports various archive tools, such as *jar* and *tar*, for submitting more than one file). The Web-CAT system will run all test cases submitted by all students against each student's solution and provides the assessment based on the results from running all test cases. For example, the solution from Student A may pass all test cases submitted by Student A, but it may not pass those submitted by Student B, thus resulting in a lower grade. Instructors may optionally provide instructor-written reference tests to ensure thoroughness or completeness of the test cases.

None of this is applicable to our parameterization framework.

8. Other Automated Assessment Tools

Some AATs, such as Aari [15], TRAKLA [16] or TRAKLA2 [17], specifically target programming exercises for data structures and algorithm courses to make sure students implement the correct algorithm. Many AATs fall into the dynamic testing category, such as Athene [18], Automatic Marker [19], BOSS [20], and Curator [21], while others such as AutoLep [22] combine both static analysis and dynamic testing. These AATs are not reviewed in detail because they do not provide any new features that may be relevant to the implementation of the parameterization framework.

B. PARAMETERIZED LABS

The Penn State University PolyLab demo system [14] is the only existing parameterized laboratory exercise reviewed and this exercise provides three demos through its website: PolyStego, PolyNet and PolyEncrypt. For each of the demos, parameterization is based on the student's email address. Based on the email address, each lab is given a unique per student configuration. The grading is done automatically when the student submits his result.

Employing the student's email address as a seed for parameterization could be useful in our framework.

C. OPENLY AVAILABLE SECURITY LAB PACKAGES

There are computer security laboratory exercises packages created by universities that are available to the public. These laboratory exercises are meant to be used by security educators to educate the public on computer security. These openly available lab packages may be useful for the iterative development of our prototype framework.

1. SEED Labs

The SEED labs are hands-on laboratory exercises for security education [1]. There are over 30 laboratory exercises in six different categories: Software Security, Network Security, Web Security, System Security, Cryptography and Mobile Security. The SEED labs provide a complete package for the student, including everything necessary for performing the laboratory exercise.

In the Software Security labs, students learn how to exploit common software vulnerabilities, such as buffer overflows, format string vulnerabilities, or the ShellShock vulnerability. In the Network Security labs, students learn about various network security technologies including firewalls, VPNs, and IPSec. Students also learn about various network attacks, such as session hijacking by exploiting TCP/IP protocol vulnerabilities. In the Web Security labs, students learn about common vulnerabilities of web-facing applications, including SQL injection attacks.

In the System Security labs and Cryptography labs, students learn by exploring various security-related technologies, such as *openssl* to learn about encryption, one-way hash functions, or role-based access control (RBAC). In the Mobile Security labs, students learn about smartphone security and perform attacks such as inserting malicious code into an existing Android application.

For the convenience of SEED labs users, a pre-built virtual machine image is provided [23]. VMWare [24] or VirtualBox [25] can be used to run those virtual machine images. For each lab, students perform the tasks as specified in the laboratory exercise description. Students capture the information necessary to answer the questions related to each task and provide a final report to the instructor for assessment.

An advantage of using the SEED labs as examples to test the prototype framework is that the SEED labs are available open source. But, there is a problem with using SEED labs. Without parameterization, previous students' reports can be used by newer students and this makes verifying whether a student indeed performed the exercise by himself difficult.

2. ITSEED Labs

The ITSEED labs are hands-on laboratory exercises for information technology security education [26]. There are a total of 12 labs in four different categories: Computer Security, Network Security, Cryptography, and Application Security. The ITSEED labs only provide the laboratory exercise descriptions. Unlike the SEED labs, no VM is provided.

For each laboratory exercise, each student must establish the required environment to perform the exercise. Students perform tasks as specified in the description for each laboratory exercise. There are questions for each task that the students must answer as part of the students' final reports to be submitted to the instructor. The labs are useful for the parameterization framework as another set of lab packages with substantial documentation.

D. VIRTUAL MACHINES AND LINUX CONTAINERS

Computer security laboratory exercises can be performed in two ways: on the bare hardware by running applications directly on the operating system, or by using virtualization.

A virtual machine (VM) presents a virtual computer, which includes the hardware instruction set. A VM can be implemented using hardware or software. One advantage of using a VM is the ability to run hardware-compatible operating systems. Virtual machines software such as VMWare or VirtualBox are widely used. VMWare is a commercial product. VirtualBox is a free and open source software.

Using virtualization technology to perform computer security laboratory exercises has several clear advantages:

1. Isolation: Using virtualization to create an environment in which to perform the exercises provides isolation, any problems that may arise from performing the exercises will be confined or isolated in the virtual host system and will not affect the actual host system.

2. Provisioning and Packaging: Exercises to be performed in a virtualized environment can be packaged and be provisioned to the students much more easily than distributing actual hardware systems.

3. Consistency: On bare hardware systems, students have to build and setup the environment to perform the exercises as specified by each exercise. Inconsistencies can occur due to differences on the actual hardware systems used by students. Packaged exercises in virtualized environment avoids the inconsistency problem because every student gets the same package.

Linux containers is an operating-system level virtualization [27]. The Linux kernel's *control group* feature bounds a collection of processes as a group. The kernel's namespace isolation feature isolates namespaces, such as process identifiers (PID), network names, and user identifiers. The Linux kernel uses these two features provide *Linux containers*. Containers limit and isolate resource usage (CPU, memory, etc.) for application processes executing within them. Containers are often compared to *chroot*, which provides "chroot jail." A process using *chroot* will have its root directory changed and subsequent operations of that process and its child processes will be limited to the new root directory and below, such that the user cannot get out of the new root directory.

Containers provide additional functionality beyond that provided by *chroot*, such as memory or disk quotas and network isolation. *Linux containers* can run on top of a Linux operating system that runs directly on top of the hardware, as shown in Figure 3, or with the Linux operating system running inside a virtual machine, as shown in Figure 4.

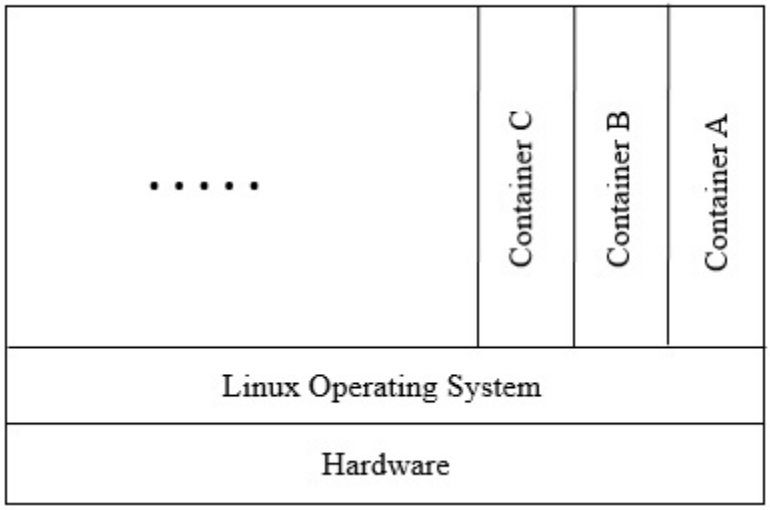

Figure 3. Containers on Linux O/S on Hardware

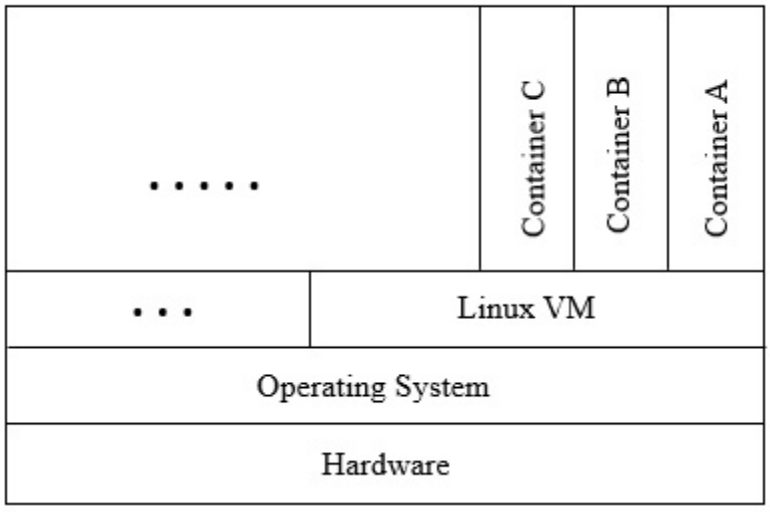

Figure 4. Containers on Linux O/S in VM

Container virtualization interfaces are available through *libvirt* [28] and LXC [29], which can be used directly to create and manage containers. Docker uses its own *libcontainer* library, in addition to *libvirt* and LXC, to provide an abstraction layer, making it easier to automate the use of containers. Images are a container abstraction, accessible through LXD [30]. An image is a static file system and a set of commands that

are to be used when the image is instantiated as a running container. Instructions that create an image are specified in a *Dockerfile*.

Docker provides interfaces for managing images and containers, such as creating and deleting images, creating a new container, starting and stopping a container, attaching and detaching from a running container. Docker also provides interfaces related to repositories, such as storing and sharing images.

Containers share the same underlying kernel as the host system, but the packages and shared libraries within a container are local to that container only. The execution environment for a given container is independently configured, with its own configuration files within its */etc/* directory. This allows for disparate Linux distributions for different containers, such that one container might run Fedora while another runs Ubuntu.

Many programming and computer security laboratory exercises are well defined and self-contained. The use of Linux containers to run or implement such exercises is appropriate. Use of containers can also provide consistency, because the containers can be installed with the libraries and tools necessary for a particular exercise. All students performing the exercise and the instructor grading the students' work will have the same containers.

The main advantage of using containers compared to using VMs is the resources required to run a container are less than those required to run a VM. A student's laptop might not be able to run multiple VMs but may be able to run many containers. This enables labs having network topologies that contain several different components. Having this flexibility makes it worth accepting the limitation that each of the containers must share the same host operating system kernel (i.e., a specific Linux kernel version).

For the prototype of the parameterization framework, Linux containers, and specifically the Docker platform, are used. Although there are other technologies that provide container-like isolation (e.g., FreeBSD Jails [31], Linux VServer [32], Solaris Containers [33], Docker is chosen for our framework. because it actively being

developed, is well maintained, and has good documentation. Docker is also very popular and widely used [34].

E. SUMMARY

This chapter discussed several AATs that were investigated as part of the background research to help identify the basic features desirable for the parameterization framework. The SEED labs were discussed. The prototype framework will make use of sample SEED labs for testing the framework's functionality. Linux containers and the reasons to use them to implement the framework were discussed.

The project description, which includes the concept of operation and the methodology used to implement the prototype framework, will be discussed in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROJECT DESCRIPTION

This chapter discusses the concept of operation for the prototype parameterization framework and the methodology used to implement the framework.

A. CONCEPT OF OPERATION

As discussed in the introduction, our framework is targeted for use on students' individual computers rather than on shared, centralized resources. The parameterization framework should be designed such that minimal resources, in terms of computer hardware, are required for the students performing the laboratory exercises and the instructor grading the students' work. To achieve this, light-weight Linux containers, specifically Docker containers, are used.

The concept of operation is depicted by the high-level view of the student's and instructor's workflows, as shown in Figure 5.

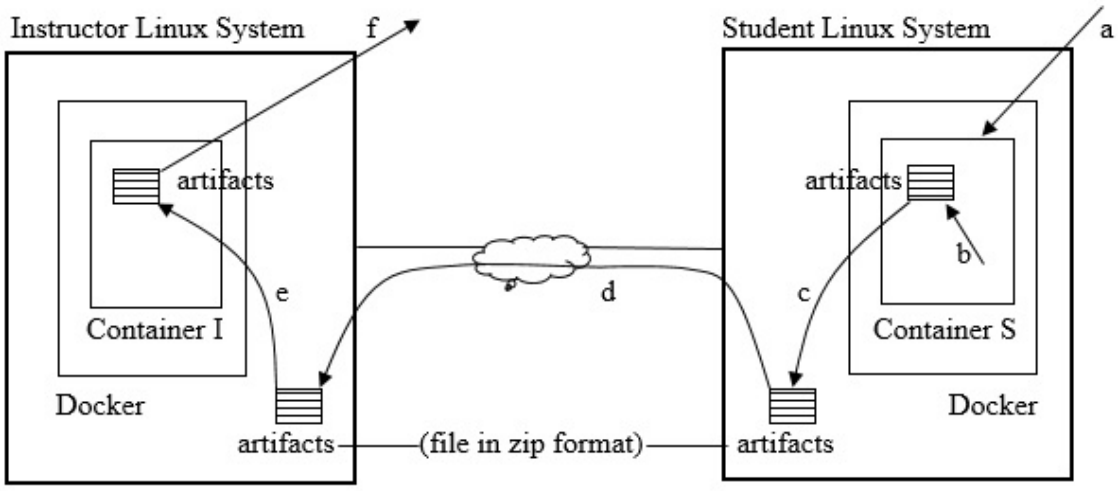

Figure 5. Student's and Instructor's Workflows

1. Students' Workflow

Students are assigned laboratory exercises by the instructor. Students download a container for the corresponding laboratory exercise (identified by step a in Figure 5).

When a student starts up his container for the first time, he will be prompted to enter his email address and the laboratory container will be parameterized based on this email address. Email addresses are used for convenience; other information agreed upon by instructor and student could also be used. Each student will perform the laboratory exercise in his own container through *bash* sessions (identified by step b in Figure 5). When the student shuts down the container, artifacts will be automatically collected in a zip file (identified by step c in Figure 5). Each student will forward his zip file to the instructor (identified by step d in Figure 5).

2. Instructors' Workflow

Instructors collect all the zip files containing the artifacts from students for a particular laboratory exercise and start the corresponding instructor container for that exercise (identified by step e in Figure 5). Instructors run the automatic assessment tool and generate a file containing the status or grade for each student (identified by step f in Figure 5). If the student's report for the assignment does not match the result or grade generated—for example, the student claims to have performed a certain task successfully, but the automatic grader reports failure to the complete the task—the instructor has the option to go back to the artifacts collected for the student to verify the student's claim of exercise completion manually.

B. METHODOLOGY

An iterative methodology is used to design and implement the parameterization framework, beginning with a simple prototype. The design process starts with simple questions regarding the intended goals for that the framework, as described in the introduction. The initial prototype of the framework starts with the basic assumption that capturing artifacts from students' containers and automatic grading will be required.

Several sample SEED labs were adapted to the framework, once the basic prototype was implemented. We started with one laboratory exercise and got it to work by revising the prototype framework as appropriate with any additional functions required. The process was repeated by adding another laboratory exercise and identifying any functionality required and revising the prototype framework as necessary. The

iterative process of revising the prototype framework and implementing sample laboratory exercises using SEED labs was repeated until the prototype framework reflected enough functionality and support for a number of possible future laboratory exercises.

There were several trade-offs and design decisions encountered during implementation of the prototype parameterization framework. When making design decisions, the effects of the decision were considered from the three different perspectives: the lab designer's, the instructor's, and the student's perspective.

Where possible, design decisions were chosen to have minimal impact on the students' learning experience. For example, student actions while performing a lab should not be altered for the purposes of collecting artifacts. The environment in which students perform the laboratory exercises should be a familiar, Linux system that is not encumbered by framework-specific menus or shell scripts. The fact that Linux containers have been used should be made known to the students only when necessary. For example, turning ASLR protection on or off on the host system impacts the students' containers.

From the lab designer's perspective, design decisions were made to simplify the expression of configuration information and avoid requiring the designer to specify redundant configuration information. The lab designer can use the parameterization framework without the need to create any additional new software unless the new software is part of the laboratory exercise.

From the instructor's perspective, implementation details of the parameterization framework are hidden. The instructor generally needs to run a simple grading script to report the results for each student. If necessary, such as when a student claims of lab completion differ from the results of the automatic grader, then the instructor has the option to reproduce the student's result based on the artifacts captured.

C. SAMPLE SEED LABS TESTED FOR PROTOTYPE FRAMEWORK

After the implementation of the initial prototype framework, sample laboratory exercises from SEED labs were used for three iterations: the "Buffer Overflow

Vulnerability Lab" and "Format String Vulnerability Lab" from Software Security labs, and "One-Way Hash Function Lab" from Cryptography labs. Since the operating system that can be supported is Linux, due to the use of Linux containers, exercises that use Minix could not be supported.

1. Format String Vulnerability Lab

For the format string vulnerability laboratory exercise, the student will learn how to exploit the format string vulnerability (such as *printf* statement) to crash a program and to read or modify an arbitrary memory location, such as internal variable.

To support the implementation of the format string vulnerability laboratory, the parameterization framework needed to provide support for capturing standard input (*stdin*) and standard output (*stdout*) when students performed the exercise. The parameterization framework also needed to provide support so that the lab exercise could be parameterized. For example, the initial value for the internal variable for each student needed to be different and thus encourage students to perform the exercise by themselves.

The parameterization framework provides the ability to capture the results or artifacts on the student's container for each operation performed by the student. The result of each operation (such as executing a program) is appropriately named, stored, and timestamped. Parsing of the captured *stdin* and *stdout* results occurs within the instructors' container. Local operations, such as running an executable program located within the student's home directory, are captured. Operations typically not considered as local (for example, running system commands such as 'ls' or 'cat') are not captured.

2. Buffer Overflow Vulnerability Lab

For the buffer overflow vulnerability laboratory exercise, the student will learn how to alter a program's execution flow by overflowing a local buffer and gaining root privilege. The original SEED laboratory exercise always presents the same buffer size to be overflowed.

Using the parameterization framework, this buffer overflow vulnerability laboratory exercise can be parameterized. For example, each student will have a

vulnerable program with a different buffer size. This change requires each student to figure out how to overflow the buffer and to modify the return address and thus alter the program's execution flow. Once the execution flow is modified and root privilege is obtained, the student will be asked to view a file accessible only by root. The content of the file will be parameterized so that each student sees a file with different contents.

To implement the buffer overflow vulnerability lab exercise, the parameterization framework had to be extended to provide a way for the instructor to assess whether a certain task in the exercise has been completed in a given circumstance. For example, a task may require the student to obtain root privilege, when the kernel has been configured so that address space layout randomization (ASLR) protection is turned on. If the student performs the task with ASLR protection turned off, the task is considered not completed. Because ALSR has to be turned on and off at the Linux host level—that is, the student must perform configuration tasks on the host—this adds minor complexity to the laboratory exercise.

3. One-Way Hash Function Lab

For the one-way hash function laboratory exercise, the student explores and becomes familiar with one-way hash functions and message authentication codes (MAC).

Using the parameterized framework, this one-way hash function laboratory exercise has been modified such that the student is asked to hash a pre-created file for which the content is different for each student. Students are asked to perform various hash function-related exercises to familiarize themselves with hash functions and MAC on a Linux system.

As part of implementing the one-way hash function lab exercise, the parameterization framework is further extended to allow the lab designer to specify criteria for exercise completion spanning multiple operations (i.e., results are contained in several files each with a different timestamp). For example, the exercise may involve having the student generate a MAC for a file, and then re-generate the MAC after the file is modified. The exercise is considered successfully completed when the MAC for the file is different and correct for each version of the file.

Implementation of this lab led us to revisit the capturing of input and output of commands issued by the student. Prior to this lab, we only captured *local operations* running an executable program stored in the student's home directory. We did not capture non-local operations (e.g., system commands, such as the "ls" or "cat" command). The framework was altered to allow the lab designer to identify arbitrary commands whose input and output are to be captured. As an example, the one-way hash function lab, is configured to capture input and output of the "openssl" command.

D. SUMMARY

This chapter discusses the concept of operation for using the parameterization framework from the point of view of the student, the instructor and the lab designer. The use of an iterative methodology was described. Sample SEED labs were adapted to test the framework, and this chapter described examples of how this led to extensions to the framework.

Implementation details will be discussed in the next chapter.

IV. IMPLEMENTATION DISCUSSION

This chapter discusses the implementation of the parameterization framework. The discussion identifies scripts implemented to manage three primary aspects of the framework: the student container, the instructor container, and laboratory exercise development support. This chapter also describes configuration files utilized by lab designers to define, parameterize, and assess student performance of lab exercises. The scripts described below include error condition checking, and the scripts will exit in the event of errors.

The description below reflects a snapshot of the development state of the framework at a particular time. The framework is an ongoing development. Script functions and configuration file syntax and semantics will evolve from the baseline described in this thesis.

A. STUDENT CONTAINER

The student containers are where each student performs specific tasks required by the instructor for corresponding lab exercise. There are two sets of functions that implement student containers: those that execute on the host to manage the creation of the container, and those performed within the container (e.g., to support the gathering of artifacts).

1. Host-Based Container Operations

This discussion assumes the student has already downloaded the container images from a Docker repository. Management of repositories is outside the scope of this thesis. The main host operations include starting the student container for the corresponding lab exercise and stopping the student container. Other supporting operations on the host include creating additional terminals for students to use when interacting with a container and pausing a running container.

a. Starting a Student Container

The script to start a student container is named “start.sh.” This script takes one command line argument specifying the lab exercise name. The script references a configuration file named “start.config.” The start.sh script uses the lab exercise name to derive the name of the container, the name of the container image, the directory name to transfer artifacts from the container and the master seed for the laboratory exercise.

The start.sh script checks to see if the container image is present on the student computer. If it is not present, it will be created (though it is intended that container images will be retrieved from a Docker Repository in future versions of the framework). The start.sh script uses Docker commands to start a container, using the file system and startup parameters defined in the image.

The first time the container for the corresponding lab exercise is started, the start.sh script will prompt the user for his email address, which is used by the start.sh script to create a laboratory instance seed. The email address and seed are stored inside the corresponding lab exercise container. The first time a container is started, the parameterization script “paramaterize.sh” (described below) is executed. The start.sh script presents the student with two virtual terminals, one displaying the corresponding lab instruction and the other a bash shell ready to accept commands (i.e., to perform the exercise specified by the instructor).

b. Stopping a Student Container

The script to stop a student container is named “stop.sh.” It takes one argument specifying the lab exercise name and references the same “start.config” file used by the start.sh script.

The stop.sh script proceeds to create the transfer directory for transferring the artifacts from the container. Prior to stopping the container, the script uses a Docker command to invoke a Python script, “Student.py,” to run inside the container to create a zip file containing the artifacts for the corresponding lab exercise (i.e., the content of the student’s home directory). The artifacts will reflect the students’ work. Students who are

unable to complete all the tasks can still submit the artifacts to the instructor for partial grading.

The stop.sh script will copy the zip file from inside the container to the transfer directory and then it will stop the container.

c. Pausing/Unpausing a Student Container

Scripts to pause and unpause a container, named pause.sh and unpause.sh, are provided for convenience. Both scripts also takes one argument specifying the lab exercise name and reference the “start.config” configuration file.

d. Additional Terminals for a Student Container

By default, the start.sh script upon successful start of a container will create two terminals for student to use. Occasionally, the student may desire more terminals, a script named moreterm.sh is provided for this reason.

2. Internal Student Container Operation

Once the student starts the container corresponding to a particular lab exercise, the student will perform the lab exercise. The first time a container is started, the start.sh script uses a Docker command to invoke the parameterization script “paramaterize.sh” which will be run within the container to parameterize the corresponding lab exercise.

a. Parameterization of a Student Container

The parameterize.sh script executed from the start.sh script, takes three arguments: the lab instance seed, the user’s email and the lab exercise name. This script stores each of these three arguments in separate files within the container for later use. This script will call a Python script “ParameterParser.py” to parameterize the lab exercise as specified by the lab designer.

This script will also check if a local script file named “fixlocal.sh” is present in the container image. This script is intended for use by the lab designer to customize the lab exercise, (for example, compiling source code to generate an executable program in 32-bit mode). If the fixlocal.sh script exists, it will be invoked.

b. ParameterParser.py script

The ParameterParser.py script does the bulk of the parameterization work. This script takes the lab instance seed and optionally a configuration file as command arguments. If no configuration file is provided, a default configuration file is assumed as "parameter.config." This script performs the parameterization of the lab exercise according to the entries specified in the parameter.config file. (as described in the lab designer section).

c. Student.py script

The Student.py script executed from the stop.sh script, uses the lab name and the user's email address previously stored inside the container to derive the name of a zip file. The script will then proceed to create a zip file containing the artifacts for the student corresponding to the lab exercise (i.e., the content of the student home directory).

d. Display Student Instruction script

The "startup.sh" script is located inside the container. It displays the content of the file named "instruction.txt" located in the home directory. This startup.sh is invoked within one of the terminals started by the start.sh script previously mentioned.

e. Bash script hooks

In order to capture the operations performed by the student without distracting the student with wrappers or background monitoring processes, the framework introduced a couple of bash scripts that hook commands issued from a bash shell. The hook checks if the command to be executed is to have its input and output recorded as part of the laboratory artifacts. Commands whose I/O are recorded include those that are local (i.e., executed from the student home directory) and those in an explicit list created by the lab designer. This list, (in the "treataslocal" configuration file), may include system commands (e.g., "ls"), which would typically be exempt from I/O capture to avoid unnecessary creation of artifact files.

If the hook determines that the command I/O is to be captured, it will call a capture script named “capinout.sh,” which is responsible for creating a copy of the standard input and standard output within a file whose names reflect the executed program.

f. Capture Artifact script

The script “capinout.sh” is responsible for creating a copy of the standard input and standard output file corresponding to the executed program. This script is designed not to interfere with the visible operation of commands, including those that include pipes. This script uses the filename of the executed program and appends “.stdin” and “.stdout” as the filenames for the files to store the standard input and standard output of the executed program.

g. checklocal.sh script

This script is created by a lab designer to capture local settings on the system along with any operations that the student performs. For example, capture the ASLR setting when the student performs buffer overflow attack.

B. INSTRUCTOR CONTAINER

The instructor container is where the instructor transfers artifacts collected from students for each laboratory exercise to perform grading. The instructor container is built to match the student container so that the instructor may reproduce the students’ results, such as when a student claims of lab completion differ from the results of the automatic grader.

As with the student container, there are two categories of scripts, those run outside the container (i.e., on the container host), and those invoked within the container.

1. Host Operations

This discussion assumes the instructor has already downloaded the container images from a Docker repository. The main host operations include starting and stopping

the instructor container. Other supporting operations on the host include creating more terminals for the instructor, and pausing an active or running container.

a. Starting an Instructor Container

The script to start an instructor container is named “start.sh.” It creates a container for use by the instructor in a manner similar to the start.sh script used to create and start student containers.

The instructor’s instance of this start.sh script will copy students’ artifacts in the form of zip files found in the transfer directory to the instructor container.

The script will then start the instructor container and present two terminals, one terminal displaying the corresponding lab instruction to the instructor and the other terminal presenting a bash shell, ready for the instructor to run the automated grading script for each student’s laboratory exercise based on the collected artifacts.

b. Stopping, Pausing and Creating Additional Terminals

The instructor is provided with stop.sh, pause.sh, unpause.sh and moreterm.sh scripts identical in function to those provided to the students.

2. Internal Instructor Container Operation

Once the instructor starts the container corresponding to a particular lab exercise, the instructor will generally run the script to automatically grade student work based on the artifacts collected for each student.

a. Instructor.py script

The automated grading script is named “Instructor.py,” and it processes the artifacts collected from the students (which were previously copied into the instructor container home directory by the start.sh script). The grading script first extracts each student zip file into its own directory. The script calls GoalsParser to parse the goals or tasks defined by the lab designer for the particular lab exercise. The script then calls ResultsParser to parse results from each student’s directory. The Grader script is called to match the goals or tasks with each student’s results. If a goal or task is completed, it will

be marked as “P”; otherwise, it will be marked as “F.” The grades for each task for each student are then stored in a resulting grade file “grades.txt.”

b. GoalsParser.py script

The GoalsParser.py script parses the goals.config configuration file (as described in the lab designer section). This script takes one argument corresponding to the directory name for each student. This script will obtain the laboratory instance seed that corresponds to each student and will call the same ParameterParser script used in the student’s container to create the parameter list for each student.

The GoalsParser.py script will validate the format for each entry line in the goals.config configuration file. Once all the goals in the configuration file are validated, the goals are stored as a JSON file named “goals.json” to be used by the automated grader script.

c. ResultsParser.py script

The ResultsParser.py script parses the results.config configuration file (as described in the lab designer section). This script takes three arguments: the directory name for each student, an instructor directory and the corresponding output filename for each student’s result.

This script validates the format for each entry line in the results.config configuration file. Each valid entry line corresponds to a result for a specific goal or task. This script parses the corresponding file stored in each student’s directory for each result. Results are tagged according to the result tag specified in the configuration file. If the result for a result tag cannot be found, then the value of that result tag will be marked as “NONE.” Once all the result tags specified in the configuration file are parsed, the results are stored as a JSON file using the filename passed in as the argument. The result file for each student will be used along with the goals.json file by the automated grader script to determine if a goal or task has been completed successfully.

d. Grader.py script

The Grader.py script takes three arguments: the directory name for each student, an instructor directory, and the lab name used to derive the output filename for each student's results.

This script uses the goals.json file that describes each goal or task for a particular laboratory exercise and determines whether that goal or task is completed based on the results file for each student.

e. evalBoolean.py script

The evalBoolean.py script is a helper script for the Grader.py script. This script contains functions to evaluate Boolean expressions that may be described as a goal.

f. Display Instructor Instruction script

Instruction files for the instructor are displayed using "startup.sh" in a manner similar to that used by students.

C. LABORATORY EXERCISE DESIGNER

This section describes configuration files and utilities utilized by lab designers to construct lab exercises within the framework.

1. Configuration Files Format

Lab designers use configuration files to define how lab exercises are parameterized for individual students, and to define criteria for evaluating student performance. The syntax and semantics of these files are described below.

a. Parameter.config

The parameter.config configuration file is used by the ParameterParser.py that performs the bulk of the parameterization work. Lines that start with "#" and empty lines will be ignored.

For each line, each token is separated by the ":" symbol. The first token on the line is the parameterization ID. The second token is defined as the operator and defines

what operations will be performed. The syntax for each token after the operator is as follows:

1. If the operator is “RAND_REPLACE,” then the entry format will be:

RAND_REPLACE : <filename> : <token> : <LowerBound> : <UpperBound>

The <filename> specifies the file that must exist inside the container where the <token> is the string to be replaced inside that file. The <LowerBound> and <UpperBound> will be used by the random generator, i.e., the lower bound and the upper bound of the value to be generated.

2. If the operator is “HASH_CREATE,” then the entry format will be:

HASH_CREATE : <filename> : <string>

The <filename> specifies the file and it will be created if it does not exist. The <string> will be used along with the user’s email as the secret keyed hash for the lab instance seed.

3. If the operator is “HASH_REPLACE,” then the entry format will be:

HASH_REPLACE : <filename> : <token> : <string>

The <filename> specifies the file that must exist inside the container where the <token> is the string to be replaced inside that file. The <string> will be used along with the user’s email as the secret keyed hash for the lab instance seed to replace the <token> string.

b. Goals.config

The goals.config configuration file is used by the GoalsParser.py. It defines each goal or task to be completed for a particular laboratory exercise. A sub-goal is an intermediate goal. Lines that start with “#” and empty lines are ignored.

There are two possible formats for each entry line:

1. <id> = <type> : <string>
2. <id> = <type> : <operator> : <resulttag> : <answertag>

The id represents the goal identification and it must consist of alphanumeric characters. If the type is “boolean,” the first format above is used and the string that follows will be evaluated as a Boolean value. The string consists of a Boolean expression naming sub-goals that have the “boolean_set” or “matchacross” type as described below.

For the second format, the type must be one of the following:

1. For “matchany,” the value corresponding to the answertag will be compared to any values corresponding to the resulttag. Note that a “matchany” goal ID will not be used as a sub-goal for goal of type “boolean.”

2. For “matchlast,” the value corresponding to the answertag will only be compared to the last value corresponding to the resulttag, i.e., using the last timestamp value for the resulttag.

3. For “matchacross,” the value corresponding to the answertag will be compared to the value of the resulttag across different timestamps. Note that a “matchacross” goal ID will not be used as a sub-goal for goal of type “boolean.”

4. A “boolean_set” type is a sub-goal to be used with goal of type “boolean.”

For the second format, the operator must be one of the following:

1. If the operator is “string_equal,” the answertag and resulttag values are treated as strings and if they are equal, the GoalsParser script sets the corresponding goal to success.

2. If the operator is “string_diff,” the answertag and resulttag values are treated as strings and if they are different, the GoalsParser script sets the corresponding goal to success.

3. If the operator is “string_start,” the answertag and resulttag values are treated as strings and if resulttag string starts with answertag string, the GoalsParser script sets the corresponding goal to success. Example: answertag value = “MySecret” and resulttag value = “MySecretSauceIsSriracha”

4. If the operator is “string_end,” the answertag and resulttag values are treated as strings and if resulttag string ends with answertag string, the GoalsParser script sets the corresponding goal to success. Example: answertag value = “Sriracha” and resulttag value = “EatMoreFoodWithSriracha”

5. If the operator is “integer_equal,” the answertag and resulttag values are treated as integers and if they are equal, the GoalsParser script sets the corresponding goal to success.

6. If the operator is “integer_greater,” the answertag and resulttag values are treated as integers and if answertag value is greater than resulttag value, the GoalsParser script sets the corresponding goal to success.

7. If the operator is “integer_lessthan,” the answertag and resulttag values are treated as integers and if answertag value is less than resulttag value, the GoalsParser script sets the corresponding goal to success.

For the second format, the resulttag must be an alphanumeric string and will be used to lookup the corresponding result value in the student’s result file.

For the answertag in the second format, we allow several different formats:

1. Look up both the answer value and the result value from the student’s result file. This is defined by expressing the answertag as either <string> or “result”.<string>

2. Compare a constant or a computed value to a value from the student’s result file. This is defined by expressing the answertag as either:

- “answer”=<string> where the <string> is the value to be compared to the value of the resulttag. Note: Characters not allowed in the <string> are “:” and “=”.

- “asciirandom”=<lowerbound>-<upperbound> where the value corresponding to the answertag is now generated using a random value generator that is seeded by the lab instance seed. The lowerbound and upperbound values are checked to make sure that they are within ASCII range. The value is chosen between the lowerbound and the upperbound and then converted to ASCII string representation.

- “hexrandom”=<lowerbound>-<upperbound> is similar to asciirandom but the value chosen is converted to a hex value.

- “intrandom”=<lowerbound>-<upperbound> is similar to asciirandom but the value chosen is converted to an integer value.

“hash”=<string> where the value for the answertag is generated by using the md5sum hash of the concatenation of the lab instance seed and the <string> value. Note: Characters not allowed in the <string> are “:” and “=”.

c. Results.config

The results.config configuration file is used by the ResultsParser.py. It identifies the expected results for each goal or task to be completed for a particular laboratory exercise. Lines that start with "#" and empty lines will be ignored.

The entry line format is as follows:

<nametag> = [*stdin* | *stdout*] : [field_type] : <field_id> : <line_type> : <line_id>

The nametag is a symbolic name for the result to be obtained from the student's artifact. The nametag must consist of only alphanumerics and underscores. This symbolic name will be referenced in the goals.config configuration file.

The *stdin* and *stdout* represent the resulting files captured when the student performs a laboratory exercise. The filename is associated with the executable program or operations that the student performs in the container.

The field_type is an optional field and the following field_type is supported:

1. field_type "TOKEN" is the default in which case the line is treated as a sequence of space-delimited tokens.
2. field_type "PARENS" is used to specify that the desired value is contained in parenthesis.
3. field_type "QUOTES" is used to specify that the desired value is contained in quotes.

The field_id is either an integer value, or the string "LAST" or "ALL." An integer is used to identify the nth occurrence of the field_type, if the field_id is "LAST," this indicates the last occurrence. If the field_id is "ALL" then this means the entire line.

The line_type identifies how to find the line in the student's artifact file. If the line_type is "LINE" then the line_id that follows must be an integer. The integer value represents the line number in the file. If the line_type is "STARTSWITH" then the line_id that follows will be a string that will be used to match the first line that starts with that string.

2. Laboratory Exercise Template

To aid the lab designer in designing laboratory exercises, a set of template files was created. The lab designer runs the “new_lab_setup.sh” script to copy a template into a new lab directory. The template includes a set of files typically needed to create a student container and an instructor container. These files include instances of the configuration files described in the previous section.

To create a student container, laboratory designers must identify any packages required in the exercise such as any particular version of libraries and modify the template Dockerfile accordingly. Any files required as part of the exercise will also be packaged in a tar-zipped file to be read into the container as specified in the Dockerfile.

The default artifacts to be collected reflect the content in the student’s home directory, which is where the student performs the laboratory exercise. Additional artifacts to be collected outside of the student’s home directory will also need to be identified by providing a link to those artifacts in the home directory. Collecting samples rather than a complete record of students’ work is identified as possible future work.

Instructor containers, typically contain all the same packages and libraries as the student container. The instructor container may have additional tools that are unique to a particular laboratory exercise and may be required for grading purposes.

3. Build Image Scripts

Once the laboratory designer has completed setting up a particular laboratory exercise directory based on the template (i.e., populating the directory with files relevant to the exercise, modifying the Dockerfile to identify the necessary packages and setting up the configuration files), both the student container and the instructor container image can be created.

Laboratory designers run the script named “buildImage.sh” to build the student container image and run the script named “buildInstructorImage.sh” to build the instructor container image. The transfer and management of these image files within a Docker Repository is outside the scope of this thesis.

D. SUMMARY

This chapter discussed the implementation details of the prototype parameterization framework. Scripts that execute in the student's container, the instructor's container and the container host were discussed. Configuration files utilized by lab designers to define, parameterize and assess labs were described. The use of a template to aid the lab designer in creating laboratory exercises was discussed.

Potential future work and conclusions are discussed in the next chapter.

V. CONCLUSION

This chapter describes possible future work that would enhance the prototype framework and the chapter includes a conclusion for this thesis

A. FUTURE WORK

Several possible enhancements identified during the implementation of the prototype framework include:

1. Multiple Containers

Laboratory exercises that consist of multiple containers would support more complex topologies such as a client and server architectures. Docker containers include support for multiple isolated networks via which containers can be interconnected. Preliminary informal testing leads us to conclude that a student laptop could host multiple containers without suffering the degradation that might occur when attempting to host multiple simultaneous VMs.

2. Multi-home Networking

Support for multiple containers could be further expanded to allow multi-home networking to reflect more realistic topologies that include a router container.

3. Trial Submissions

Students might benefit from a capability to submit their work on a trial basis (e.g., to functions contained within the student's own computer), to get immediate feedback on how he or she has performed so far.

4. Artifact Collection Options

The artifact collection implemented in the prototype parameterization framework is the content of the student's home directory. Additional artifacts outside of the student's home directory are collected by creating a link to those additional artifacts in the

student's home directory. A future extension might allow the lab designer to identify files at arbitrary locations within the container.

There may be occasions when it is not necessary to collect everything generated in the course of a laboratory exercise, such as situations where the exercise creates extremely large files. An option to collect selected samples may be a useful feature.

5. Snapshots

Students may want to save the current state of their containers (i.e., create snapshots). This would allow students to continue working from a previously saved state, and recover from a mishap, such as after inadvertently deleting files.

B. CONCLUSION

In the introduction, we set out to identify the kinds of automated support that might assist the designer of laboratory exercises to achieve the following:

1. Determine that students performed lab exercises, with some identification of problem areas, specifically, to easily determine if there are portions of a lab that many students struggle with.
2. Gain confidence that the students did their own work, and did not obtain their exercise results from other students or the Internet;
3. Provide all students with an identical environment in which to conduct the lab exercise to help ensure that student failures and frustrations are not due to administrative and configuration problems.

Although there are many existing AATs, none provide features that support the intended goals we set out to achieve. With the implementation of the parameterization framework prototype, we provide the following:

1. A default assessment tool to help assess the students' work and potentially help to identify problem areas with laboratory exercises.
2. A mechanism for parameterizing laboratory exercises such that students have to perform their own work.
3. A consistent environment for the students and instructors by using the Linux containers.

This framework is also built such that instructors or lab designers can add additional features to the framework as needed.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SOURCE CODE

The source code for this project is kept under source code control in Subversion repository, and is maintained internally at Naval Postgraduate School at https://tor.ern.nps.edu/svn/proj/seed/. The source code listing is described in the implementation chapter and is kept under the following directory structure:

trunk/

scripts/

designer/

bin/

templates/

bin/

config/

dockerfiles/

instr_config/

MyInstructorDocker/

bin/

config/

MyStudentDocker/

bin/

config/

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. DOCUMENTATION

Documentation is also kept under source code control, under the same Subversion repository https://tor.ern.nps.edu/svn/proj/seed/. Manuals including the lab designer guide to help laboratory designers that intend to create or adapter his own laboratory exercises to use the parameterization framework are kept under the following directory structure:

trunk/

docs/

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SAMPLE LABORATORY EXERCISES

This appendix discusses the three sample laboratory exercises from SEED labs that were used during the implementation of the prototype framework.

A. FORMAT STRING VULNERABILITY EXERCISE

For the format string vulnerability laboratory exercise, the student will learn how to exploit the format string vulnerability (such as *printf* statement) to crash a program and to read or modify an arbitrary memory location, such as an internal variable.

To perform the exercise, the student runs a vulnerable program, appropriately named "vul_prog." The program prompts the student to input a decimal integer and a string. Due to the use of *printf* in a vulnerable way in the program, the student's input may cause the program to crash, and may cause the program to display or modify a variable internal to the program.

1. Source Code Change

The source code for the vulnerable program "vul_prog.c" has been modified from the original SEED laboratory exercise. The original value for an internal variable was set to SECRET2 and SECRET2 is defined to be 0x55. When the student is asked to display this value, the original program will always display the value 0x55.

In the modified version of the vulnerable program, the SECRET2 value is now set to the string "SECRET2_VALUE," which will be parameterized for each student's container according to the parameter.config configuration file.

2. parameter.config configuration file

The parameter.config configuration file contains the following line:

```
SECRET2 : RAND_REPLACE : /home/ubuntu/vul_prog.c : SECRET2_VALUE : 0x41 : 0x5a
```

The entry above tells the parameterization script to replace the string "SECRET2_VALUE" with a random value chosen from the range 0x41 to 0x5a. The random value generator is seeded with each student's laboratory exercise seed.

3. results.config configuration file

The results.config configuration file contains the following lines:

```
crashStringCanary = vul_prog.stdout : 3 : STARTSWITH : *** stack smashing detected

crashStringSignal = vul_prog.stdout : 3 : STARTSWITH : program exit,

origsecret1value = vul_prog.stdout : 6 : LINE : 8

newsecret1value = vul_prog.stdout : 6 : STARTSWITH : The new secrets:

leaked_secret1 = vul_prog.stdout : LAST : LINE : 7
```

The vul_prog.stdout indicates the *stdout* file captured when the student runs the vul_prog program. The first entry tells the result parser to parse for a line that starts with the string "*** stack smashing detected" and to store the third token as crashStringCanary. The second entry tells the result parser to parse for a line that starts with the string "program exit" and to store the third token as crashStringSignal. The third entry tells the result parser to parse and store the sixth token from the eighth line as origsecret1value. The fourth entry tells the result parser to parse for line that starts with the string "The new secrets:" and store the sixth token as newsecret1_value. The last entry tells the parser to parse and store the last token in the seventh line as leaked_secret1.

All the stored values are used to grade the student based on the collected artifacts from the student.

4. goals.config configuration file

The goals.config configuration file contains the following lines:

```
_crash_smash = boolean_set : string_equal : crashStringCanary : answer=smashing

_crash_sig = boolean_set : string_equal : crashStringSignal : answer=segmentation

crash =  boolean : ( _crash_smash or _crash_sig )

leaked_secret = matchany : string_end : leaked_secret1 : parameter_ascii.SECRET2

modifyvalue = matchany : string_diff : newsecret1value : result.origsecret1value

modifyspecific = matchany : string_equal : newsecret1value : answer=0xa
```

The first entry tells the goals parser to compare the crashStringCanary value with the string “smashing” and set the Boolean value of _crash_smash to correspond to whether the two strings are equal or not. The second entry tells the goals parser to compare the crashStringSignal value with the string “segmentation” and set the Boolean value of _crash_sig to correspond to whether the two strings are equal or not. The third entry represents the goal of crashing the program, and it is determined by the previous two Boolean values _crash_smash or _crash_sig. If either value is true, the goal is considered achieved, i.e., the student has successfully crashed the program.

The fourth entry is a goal entry that specifies that if leaked_secret1 value treated as a string, and that string ends with the string that matches the SECRET2 value set in the parameter.config configuration file (i.e., the random string set to replace the SECRET2_VALUE in the vulnerable program).

The fifth entry is a goal entry that specifies that if the newsecret1value is different from the origsecret1value, it means that the goal of modifying the value has been achieved by the student.

The last entry is a goal entry that specifies that if the newsecret1value is equal to the string 0xa, then the goal of modifying an internal variable to a specific value has been achieved by the student.

B. BUFFER OVERFLOW VULNERABILITY EXERCISE

For the buffer overflow vulnerability laboratory exercise, the student will learn how to alter a program’s execution flow by overflowing a local buffer and gaining root privilege.

To perform the exercise, the student runs an exploit program that creates a file containing a shell code. The student runs the vulnerable program named “stack” that will read the file created into a buffer. If the buffer is overflowed correctly, execution flow will transfer to the shell code and root privilege is obtained. The original SEED laboratory exercise always presents the same buffer size to be overflowed in the vulnerable program.

In the original SEED laboratory exercise, the student is asked to do the buffer overflow when the ASLR is turned off and also when the ASLR is turned on. When ASLR is turned on, the attack becomes much harder because the kernel changes the address of the entire stack. Most of the time, the hard coded return address stored in the file created using the exploit program will no longer match. The original SEED exercise tasks the student to run a simple while loop that repeatedly calls the program until it finally gains the root privilege because the return address happens to match.

To simplify the task when the ASLR is turned on, a simple bash script is now provided to the student and this script perform the while loop.

1. Source Code Change

The source code for the vulnerable program “stack.c” and the exploit program “exploit.c” are modified from the original SEED laboratory exercise. The original “stack.c” program contains a buffer that is always set to the value of 24 and the file that will contain the shell code that will overflow the buffer is always set to 517.

The hard-coded values are now modified as BUFFER_SIZE and OVERFLOW_SIZE respectively and they will be parameterized for each student’s container according to the parameter.config configuration file.

2. parameter.config configuration file

The parameter.config configuration file contains the following line:

rand1 : RAND_REPLACE: /home/ubuntu/stack.c : BUFFER_SIZE : 100 : 500

rand2 : RAND_REPLACE: /home/ubuntu/stack.c : OVERFLOW_SIZE : 1000 : 1000

rand3 : RAND_REPLACE: /home/ubuntu/exploit.c : OVERFLOW_SIZE : 1000 : 1000

roothash : HASH_REPLACE : /root/.secret : ROOT_SECRET : mysupersecretrootfile

The rand1 entry tells the parameterization script to replace the string “BUFFER_SIZE” in the “stack.c” file with a random value chosen from the range 100 to 500 in the “stack.c” file. The rand2 entry and the rand3 entry tell the parameterization script to replace the string “OVERFLOW_SIZE” in both “stack.c” and “exploit.c” with

the value 1000. The last entry tells the parameterization script to replace the string "ROOT_SECRET" in the file ".secret" stored in the root directory with a hash value of the laboratory instance seed concatenated with the string "mysupersecretrootfile." The random value generator is seeded with each student"s laboratory exercise seed. Since each student's laboratory exercise seed is different, the random value generator will generate different values for each student.

3. results.config configuration file

The results.config configuration file contains the following lines:

rootsecret = stack.stdout : 6 : STARTSWITH : My ROOT secret string is:

aslr_setting = checklocal.stdout : 3 : STARTSWITH : kernel.randomize_va_space

whilesecret = whilebash.sh.stdout : 6 : STARTSWITH : My ROOT secret string is:

The stack.stdout indicates the *stdout* file captured when the student ran the vulnerable stack program. The checklocal.stdout corresponds to the checklocal.sh script. The whilebash.sh.stdout corresponds to the simple bash script provided to the student to do the simple while loop.

The first entry tells the result parser to parse for line that starts with the string "my ROOT secret string is" in the stack.stdout file and store the sixth token as rootsecret. The second entry tells the result parser to parse for line that starts with the string "kernel.randomize_va_space" in the checklocal.stdout file and store the third token as aslr_setting. The third entry tells the result parser to parse for line that starts with "My ROOT secret string is" in the whilebash.sh.stdout file and store the sixth token as whilesecret.

All the stored values are used to grade the student based on the collected artifacts from the student.

4. goals.config configuration file

The goals.config configuration file contains the following lines:

gainrootprivilege = matchany : string_equal : rootsecret : parameter.roothash

_aslron = boolean_set : integer_equal : aslr_setting : answer=2

_looproot = boolean_set : string_equal : whilesecret : parameter.roothash

whilegetroot = boolean : (_aslron and _looproot)

The first entry tells the goals parser to compare the rootsecret value with the roothash set in the parameter.config configuration file, i.e., the hash string that replaces "ROOT_SECRET" in the file ".secret" stored in the root directory. This goal entry specifies that if the comparison of these two strings shows that they are the same, then the student has achieved a root privilege since the student is able to display the contents of a specific file stored in the root directory.

The second entry tells the goals parser to compare the aslr_setting value with the integer value of 2 and set the Boolean value of _aslron to correspond to whether the two integers are equal or not.

The third entry tells the goals parser to compare the whilesecret value with the same roothash set in the parameter config. Then the GoalsParser script sets the Boolean value of _looproot to correspond to whether the two strings are equal or not.

The last entry represents the goal of achieving root privilege while the ASLR is turned on. This goal uses the previous two Boolean values _aslron and _looproot. If both values are true, the goal is considered to have been achieved, i.e., the student has successfully displayed the content of a file stored in the root directory while the ASLR is turned on.

C. ONE-WAY HASH FUNCTION LAB

For the one-way hash function laboratory exercise, the student will learn how explore and get familiar with one-way hash functions and message authentication codes (MAC). Students also learn about the one-way property of hash functions.

To perform the exercise, the student experiments with running *openssl* command with various options. The student is asked to hash a pre-created file for which the content is different for each student. Students are asked to perform various hash function-related exercises to familiarize themselves with hash functions and MAC on a Linux system.

1. Source Code Change

This laboratory exercise does not involve students writing programs. Instead students learn by exploring use of the *openssl* command.

2. parameter.config configuration file

The parameter.config configuration file contains the following line:

DIGESTFILE : HASH_REPLACE : /home/ubuntu/filetodigest.txt : DIGEST_SECRET : mydigestsecretubuntufile

This entry tells the parameterization script to replace the string "DIGEST_SECRET" in the file "filetodigest.txt" stored in the user's home directory with a hash value of the laboratory instance seed concatenated with the string "mydigestsecretubuntufile." The random value generator is seeded with each student's laboratory exercise seed.

3. results.config configuration file

The results.config configuration file contains the following lines:

md5filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : MD5

sha1filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : SHA1

sha256filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : SHA256

hmacmd5filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : HMAC-MD5

hmacsha1filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : HMAC-SHA1

hmacsha256filedigest = openssl.stdout : PARENS : 1 : STARTSWITH : HMAC-SHA256

hmacsha256digest = openssl.stdout : 2 : STARTSWITH : HMAC-SHA256

The openssl.stdout indicates the *stdout* file captured when the student runs the *openssl* program. The first six entries have the same format, each tells the result parser to search for a line that starts with the string for each operation performed by the student using *openssl* with the appropriate options, i.e., MD5, SHA1, SHA256, HMAC-MD5,

HMAC-SHA1 and HMAC-SHA256. The line is parsed using parenthesis as a token separator and the first token is stored accordingly.

The last entry tells the result parser to parse for line that starts with the string HMAC-SHA256 and store the second token as hmacsha256digest.

Notice the difference between the sixth and the last line, the sixth line will obtain the filename that *openssl* operated on, whereas the last line will obtain the hash value.

All the stored values are used to grade the student based on the collected artifacts from the student.

4. goals.config configuration file

The goals.config configuration file contains the following lines:

```
md5done = matchany : string_equal : md5filedigest : answer=filetodigest.txt
sha1done = matchany : string_equal : sha1filedigest : answer=filetodigest.txt
sha256done = matchany : string_equal : sha256filedigest : answer=filetodigest.txt
hmacmd5done = matchany : string_equal : hmacmd5filedigest : answer=filetodigest.txt
hmacsha1done = matchany : string_equal : hmacsha1filedigest : answer=filetodigest.txt
hmacsha256done = matchany : string_equal : hmacsha256filedigest : answer=filetodigest.txt
hmacsha256diff = matchacross : string_diff : hmacsha256digest : hmacsha256digest
```

The first six entries have the same format and tell the goals parser to compare the corresponding digest value stored with the string value filetodigest.txt. These goals confirm that the student performs hash digest on the filetodigest.txt stored in the user's home directory.

The last entry tells the goals parser to match the value stored in hmacsha256digest across different timestamps and the goal is considered achieved if there exists a pair of values that have different strings. This goal is intended to confirm that the student created a hash digest for two slightly different versions of the same file. The second version contains a small modification, so the two hashes are different.

LIST OF REFERENCES

[1] SEED Labs (n.d.). SEED Labs – Syracuse University. [Online] http://www.cis.syr.edu/~wedu/seed/. Accessed Dec 10, 2016.

[2] Linux Containers (n.d.). Linux Containers. [Online] https://linuxcontainers.org. Accessed Dec 10, 2016.

[3] What is Docker (n.d.). Docker Inc. [Online] https://www.docker.com/what-docker/. Accessed Dec 14, 2016.

[4] R. S. Pettit, J. D. Homer, K. M. Holcomb, N. Simone, and S. A. Mengel, "Are automated assessment tools helpful in programming courses," American Society for Engineering Education. *122nd ASEE Annual Conference & Exposition*, 2015, 10.18260/p.23569.

[5] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments" in *Computer Science Education*, vol. 15, no. 2, pp. 83–102.

[6] P. Nordquist, "Providing accurate and timely feedback by automatically grading student programming labs." in *J. Comput. Sci. Coll* 23, 2 (December 2007), pp. 16–23.

[7] C. Higgins, T. Hergazy, P. Symeonidis, and A. Tsinsifas, "The CourseMarker CBA system: improvements over Ceilidh." in *Education and Information Technologies* 8(3), September 2003, pp. 287–304.

[8] S. Benford, E. Burke, E. Foxley, and C. Higgins, "The Ceilidh system for the automatic grading of students on programming courses" in *Proceedings of the 33rd annual on Southeast regional conference* (ACM-SE 33), pp. 176–182.

[9] Automatic Programming Assignment Checker (n.d.). BitBucket Repository. [Online] https://bitbucket.org/frantiseks/apac. Accessed Dec 14, 2016.

[10] GROK Learning (n.d.). GROK Learning. [Online] https://groklearning.com/. Accessed Dec 15, 2016.

[11] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, San Jose, CA USA, February 27–March 1, 1997, pp. 335–339.

[12] VMChecker–Automatic assignment checker (n.d.) GitHub Repository. [Online] https://github.com/rosedu/vmchecker. Accessed Dec 21, 2016.

[13] Web-CAT – The web-based center for automated testing (n.d.). GitHub Repository. [Online] https://github.com/web-cat. Accessed Dec 21, 2016.

[14] PolyLab Demo System (n.d.). Pennsylvania State University. [Online] http://polylab.ist.psu.edu:8080/demo/. Accessed Jan 10, 2017.

[15] A. Taherkhani, A. Korhonen, and L. Malmi, "Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course" in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (Koli Calling '12), pp 83–92.

[16] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander, "Experiences on automatically assessed algorithm simulation exercises with different resubmission policies" in *Journal on Educational Resources in Computing* (JERIC), vol. 5, issue 3, September 2005, article no. 7.

[17] A. Korhonen, L. Malmi, and P. Silvasti, "TRAKLA2: A framework for automatically assessed visual algorithm simulation exercises" in *Proceedings of the Third Annual Baltic Conf. on Computer Science Education*, 2003, Joensuu, Finland, pp. 48–56.

[18] D. Towell, and B. Reeves, "From walls to steps: using online automatic homework checking tools to improve learning in introductory programming courses," presented at *Association for Computer Educators in Texas* (ACET) *Journal of Computer Education and Research*, 2010.

[19] H. Suleman, "Automatic marking with Sakai" in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology* (SAICSIT '08), pp. 229–236.

[20] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *Journal on Educational Resources in Computing* (JERIC), vol. 5, issue 3, September 2005, article no. 2.

[21] S. Edwards, "Improving student performance by evaluating how well students test their own programs," *Journal on Educational Resources in Computing* (JERIC), vol. 3, issue 3, September 2003, article no. 1.

[22] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, "Ability-training-oriented automated assessment in introductory programming course," *Computers & Education*, vol. 56, issue 1, January 2011, pp. 220–226.

[23] SEED Labs: Pre-built virtual machine images (n.d.). Syracuse University. [Online] http://www.cis.syr.edu/~wedu/SEEDUbuntu12.04.zip. Accessed Jan 21, 2017.

[24] VMWare (n.d.). VMWare Inc. [Online] http://www.vmware.com/. Accessed Jan 21, 2017.

[25] VirtualBox (n.d.). VirtualBox. [Online] https://www.virtualbox.org/wiki/VirtualBox/. Accessed Jan 22, 2017.

[26] ITSEED: Active-learning laboratory experiments for IT security education (n.d.) Michigan Technological University. [Online] http://www.ece.mtu.edu/~xinlwang/itseed/index.html. Accessed Feb 12, 2017.

[27] Y. Yu, "OS-level virtualization and its applications," Ph.D. dissertation, Dept. Comp. Sci., Stony Brook Univ., Stony Book, NY, 2007.

[28] Libvirt (n.d.). Libvirt virtualization API. [Online] https://libvirt.org/. Accessed Mar 1, 2017.

[29] What's LXC? (n.d). Linux Containers - LXC [Online] https://linuxcontainers.org/lxc/. Accessed Jan 22, 2017.

[30] What's LXD? (n.d). Linux Containers - LXD [Online] https://linuxcontainers.org/lxd/. Accessed Jan 22, 2017.

[31] P. H. Kamp, and R. Watson, "Jails: confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, 2000, pp. 1–15.

[32] Linux-vserver technology (n.d). Linux VServer. [Online] http://linux-vserver.org/Overview. Accessed Jan 22, 2017.

[33] Oracle Solaris Containers (n.d.). Oracle Inc. [Online] http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html. Accessed Jan 22, 2017.

[34] Docker community passes two billions pulls (n.d.). Docker Inc. [Online] https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/. Accessed Feb 1, 2017.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California