SEED: A Suite of Instructional Laboratories for Computer SEcurity EDucation*

Wenliang Du, Zhouxuan Teng, and Ronghua Wang
Department of Electrical Engineering and Computer Science
Syracuse University. Syracuse, New York 13244
wedu@ecs.syr.edu, zhteng@syr.edu, rwang01@ecs.syr.edu

ABSTRACT

To provide students with hands-on exercises in computer security education, we have developed a laboratory environment (SEED) for computer security education. It is based on VMware, `Minix`, and `Linux`, all of which are free for educational uses. Based on this environment, we have developed ten labs, covering a wide range of security principles. We have used these labs in our three courses in the last four years. This paper presents our SEED lab environment, SEED labs, and our evaluation results.

Categories and Subject Descriptors

K.3.2 [**Computer & Information Science Education**]: Computer Science education

General Terms

Computer Security

Keywords

Security, laboratory, instructional operating system

1. INTRODUCTION

The importance of experiential learning has long been recognized in the learning theory literature. Lewin (1951) claimed that learning is attained through active participation in the learning process; and Piaget (1952) stated that learning occurs as a result of the interaction between the individual and the environment [7]. Computer scientist Denning (2003) also indicated that if we adopt a picture that ignores practice, our field (computing) will end up like the failed "new math" of the 1960s–all concepts, no practice, lifeless; dead[4].

The importance of experiential learning has also been recognized in computer science education. Traditional courses, such as Operating Systems, Compilers, and Networking, have effective laboratory exercises that are widely adopted. Unfortunately, computer security education, which is still at its infancy, has yet had widely-adopted laboratory exercises. Although there does exist a small number of good individual laboratories [5, 6, 8], their coverage on security principles is quite narrow; many important security principles, concepts, and innovative ideas are not covered by the existing laboratories. Moreover, since these existing laboratories are developed by different people, they are built upon a variety of different lab environments (operating systems, software, etc.). Usually, there is a steep learning curve to learn a new lab environment. If an instructor wants to use several laboratories, students need to learn a few different environments, which is impractical for a semester-long course.

To fill the aforementioned void in security education, we propose *a general laboratory environment and a comprehensive list of laboratory activities that are essential to security education*. We call our laboratory environment the *SEED* environment (SEcurity EDucation), and each laboratory activity a *lab*. SEED has a number of appealing properties: (1) *SEED is a general environment*. It supports a variety of labs that cover a wide spectrum of security concepts and principles, as well as important security engineering skills. It provides a common laboratory environment for computer security education. (2) *SEED is based upon a proven pedagogic method*. The core of the SEED environment consists of an instructional operating system. Using instructional operating systems has proven to be effective in traditional computer science courses, such as Operating System, Networking, etc [2, 3, 9]. This project is the first to apply such a pedagogical method in security education. (3) *SEED is a low-cost environment*. It can be easily set up on students' personal computers using free software.

Based on the SEED environment, we have developed ten labs for computer security education. During the last four years, we have used these labs in three different courses, including *Internet Security*, *Computer Security*, and *Introduction to Computer Security*. The first two are graduate courses, and the last one is a undergraduate course. We have conducted evaluation after students finish each lab. The results are quite encouraging. In this paper, we describe the SEED lab environment, lab setup, and various practical issues (Section 2). We also describe a suite of SEED labs that

*This work is supported in part by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. The lab materials from this work can be obtained from http://www.cis.syr.edu/∼wedu/seed/.

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

we have developed, along with our experience with them (Section 3). Finally, we report our evaluation results.

2. SEED LAB ENVIRONMENT

The design goal of our labs is intended to ask students to conduct one or a few of the following tasks in each lab: (1) explore (or "play with") an existing security component of the OS, (2) modify an existing security component, (3) design and implement a new security functionality, (4) test a security component, and (5) identify the vulnerabilities in the OS, and exploit such vulnerabilities. Some of these tasks can be conducted without looking at the source code, and thus, can be conducted in most of the modern operating systems. However, some tasks, such as tasks 2, 3, and perhaps 4 (for white-box testing), do require code reading and/or code writing; using a production operating system (e.g. `Linux` and `Windows`) for these tasks is not an effective approach. Most of the security components that students need to address are not stand-alone components, they interact with various other components within the OS, such as file system, process management, memory management, and network services. Students need to understand these interactions in order to conduct the tasks. Understanding these interactions in a production operating system is not a feasible task during a single semester for average students.

This challenge is not unique in security education, similar problems have been faced by instructors of traditional computer science courses when they chose platforms for their course projects. A successful pedagogical approach emerged from the practice; that is, the use of instructional systems. For example, Operating System and Networking courses often use instructional operating systems, such as `Minix` [9] `Nachos` [2], and `Xinu` [3]; compiler courses often use instructional languages such as `miniJava` [1]. These systems are designed solely for educational purposes. Unlike the production systems, they do not have many fancy features; they only maintain the components that are essential for their designated educational purpose. As a result, rather than having millions of lines of code as many production operating systems do, instructional OSes only have tens of thousands of lines of code. Understanding the interaction of the components in these systems has proven to be feasible in the educational process of the traditional computing courses [2, 3, 9]. The use of instructional systems allows students to apply knowledge and skills to a manageable system, thus allowing them to remain active participants in the learning process.

To achieve our design goals, we run two types of OSes in the SEED environment. One type runs `Minix`, an instructional operating system. For the labs that require students to read, understand, and modify source code, we use `Minix` as the platform because supporting coding-related tasks is the strength of the instructional OSes. In the SEED environment, we also use a production OS (we chose `Linux`). Production OSes provide a rich set of security systems. We use them primarily for exploration-type labs, in which students play with a security system to learn how things work and how security breaches can occur. Students do not need to read or modify the source code of these operating systems.

Virtual Machines: To be able to run `Minix` and `Linux` (sometimes multiple instances of them) conveniently in a general computing environment, we use the virtual machine technologies. Students can create "virtual computers" (called *guest* computers) within a physical computer (called *host* computer). The host computer can be a general computing platform, while each guest computer can run its own operating system, such as `Minix` and `Linux`. The guest computers and the host computer can form virtual networks. All of this can be accomplished using virtual machine softwares, such as VMware and Virtual PC. VMware has recently established an academic program, which makes the license of all VMware software free for educational uses. Since `Minix` and `Linux` are also free, the software cost for establishing SEED environment is zero.

Due to the virtual machine technologies, the SEED lab environment does not require a physical laboratory. Students can create the entire SEED environment on their personal computers. We did a survey in our classes; 85% of our students actually prefer to use their own computers for their lab assignments, while another 15% feel ok to use their own computers, but prefer to use public computers for the labs. An alternative is to install VMware on the machines in public laboratories. However, since each virtual machine needs 300 MB to 1 GB disk space, this approach creates a high demand on disk space on public machines, which is impractical in many institutions. There is an easy solution to this problem: just ask students to buy a portable hard-disk (cost less than $100). They can store their virtual machines on the portable hard-disks, and work on their lab assignments on any public machines with VMware installed.

3. SEED LABS

Based on the SEED environment, we have developed two types of labs: *implementation labs* and *exploration labs*. (1) The objective of the implementation labs is to provide students with opportunities to apply security principles in *designing and implementing* systems. Since coding is necessary for these labs, we use `Minix` as the platform. (2) The objective of the exploration labs is two-fold: the first is to enhance students' learning via observation, playing and exploration, so they can see what security principles "feel" like in a real system. The second objective is to provide students with opportunities to apply security principles in *analyzing and evaluating* systems. The target systems for students to explore include the systems that come with the underlying operating systems (both `Minix` and `Linux`) and the systems that we build in the implementation labs. We have developed and used 10 different labs in our classes. Due to page limitation, we only describe three selected labs in detail, while giving brief summaries to the others.

3.1 `Set-UID` Lab

The learning objective of this lab is for students to investigate how the `Set-UID` security mechanism works in `Minix`, discover how program flaws in `Set-UID` programs can lead to system compromise, and identify how modern operating systems protect against attacks on vulnerable `Set-UID` programs. `Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program is run, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but unfortunately, it is also the culprit of many bad things.

In this exploration lab, students' main task is to "play"

with the Set-UID mechanism in Minix and Linux, report and explain their discoveries, analyze why some Set-UID behaviors in Linux are different from that in Minix. In particular, they need to accomplish the following tasks:

Task 1 (Understanding Set-UID). Figure out why chsh, passwd, and su commands need to be Set-UID programs. Using the Minix source code, figure out how Set-UID is implemented in the OS, and how it affects the access control.

Task 2 (Potential risks of Set-UID programs) The library function system(const char *cmd) can be used to execute a command within a program. The way system(cmd) works is to invoke the /bin/sh program, and then let the shell program to execute cmd. Because of the shell program invoked, calling system() within a Set-UID program is extremely dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as PATH; these environment variables are under user's control. By changing these variables, malicious users can control the behavior of Set-UID programs.

To see how this type of attack works, students are given a Set-UID program that simply calls system("ls"). This program is supposed to execute the /bin/ls command, but the programmer "forgets" to use the absolute path for the ls command. Students need to find ways to "trick" this Set-UID program into running another command instead of /bin/ls. Students need to observe whether the new command is executed with the root privilege. Students need to conduct the tasks in both Linux and Minix.

Task 3 (An alternative). To convince students to never use system(), and instead to use execve(), two Set-UID programs are given to students. The first one simply calls system("ls"), and the second one replaces system() with execve(), the one that does not invoke a shell program. Students need to run both programs in Minix and Linux, and explain their observations.

Task 4 (Another potential risk). To be more secure, Set-UID programs usually invoke the setuid() system call to permanently relinquish their root privileges. However, this is not enough in certain circumstance. For example, when the program has already opened a file (e.g. the password file) with root privileges, after the program drops its root privileges, it can still access that file because the file descriptor is still valid. Therefore, although the privileges are already dropped, systems can still be compromised. To demonstrate this, we give students a Set-UID program (omitted due to page limitation), ask them to run it, and then explain their observations.

Experience: This lab takes one to two weeks to finish. Students were very interested in this lab; in particular, they were intrigued by the difference between Minix and Linux. For example, in Task 2, students were first surprised to see that Linux, unlike Minix, was immune to the attack. Many students were curious about that, and they tried very hard to find out why Linux was protected. They investigated various hypothesis. Eventually, they traced the protection to the shell program /bin/sh, which by default, automatically dropped the Set-UID privilege when invoked. There was a lot of discussion in the class on this protection measure. From this lab, the best lesson we have learned is the benefit of "comparison studies". Minix, an instructional OS not specifically designed for security courses, is a much less secure OS than Linux; there are many other security-related differences between these two OSes. We will continue using this comparison strategy to develop our future labs.

3.2 Capability Lab

The learning objective of this lab is for students to apply the capability concept to enhance system security. In Unix, there are a number of privileged programs (e.g., Set-UID programs); when they are run by any user, they run with the root's privileges. Namely the running programs possess all the privileges that the root has, despite of the fact that not all of these privileges are actually needed for the intended tasks. This design clearly violates an essential security engineering principle, the *principle of least privilege*. As a consequence of the violation, if there are vulnerabilities in these programs, attackers might be able to exploit the vulnerabilities and abuse the root's privileges.

Capability can be used to replace the Set-UID mechanism. In Trusted Solaris 8, root's privileges are divided into 80 smaller capabilities. Each privileged program is only assigned the capabilities that are necessary, rather than given the root privilege. A similar capability system is also developed in Linux. In a capability system, when a program is executed, its corresponding process is initialized with a list of capabilities (tokens). When the process tries to access an object, the operating system should check the process' capability, and decides whether to grant the access or not. In this lab, students need to implement a simplified capability system for Minix.

Required Capabilities: To make this lab accomplishable within a short period of time, we have only defined 5 capabilities. Due to our simplification, these five capabilities do not cover all of the root's privileges, so they cannot totally replace Set-UID. They can only be used for privileged programs that just need a subset of our defined capabilities. For those programs, they do not need to be configured as a Set-UID program; instead, they can use our capability system. Our system supports the following capabilities:

1. CAP_READ: Allow read on files and directories. It overrides the ACL restrictions regarding read on files and directories.

2. CAP_CHOWN: Overrides the restriction of changing file ownership and group ownership.

3. CAP_SETUID: Allow to change the effective user to another user. Recall that when the effective user id is not root, callings of setuid() and seteuid() to change effective users are subject to certain restrictions. This capability overrides those restrictions.

4. CAP_KILL: Allow killing of any process. It overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.

5. CAP_SYS_REBOOT: Allow rebooting the system.

We *intentionally* made the above description of capabilities vague. If not carefully designed, a system that follows this vague description might have loopholes. For example, more restriction must be put on the CAP_SETUID capability; otherwise, a process with such a capability can gain all

the other capabilities. Students are given the responsibility to identify potential loopholes in the above description and further clarify the description.

Managing Capabilities: A process should be able to manage its own capabilities. For example, when a capability is not needed in a process, the process should be able to permanently or temporarily remove this capability. Therefore, even if the process is compromised, attackers are still unable to gain the privilege. In this lab, students need to support a list of standard functionalities, including (temporarily) `Disabling/Enabling`, (permanently) `Deleting`, `Delegating`, and `Revoking` capabilities.

Experience: Before we used this lab in our class, we predicted that students might have trouble figuring out how the system calls work in `Minix`, and how different components of `Minix` exchange data (`Minix` is a micro-kernel operating system, it uses messages for components to exchange data). We have developed corresponding documents to help students. Students have found these documents extremely useful. However, we failed to predict another difficulty: students spent a lot of time on figuring out how to store information in `i-nodes`. We decided that this task was not essential to computer security. Therefore, we have developed another document to describe detailed instructions on how to manipulate the `i-node` data structure.

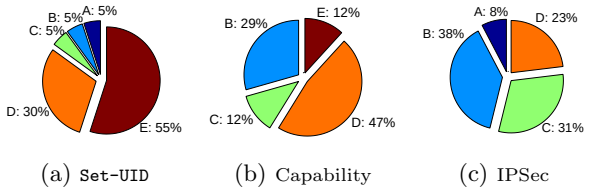
3.3 IPSec Lab

The learning objective of this lab is for students to integrate a variety of security principles to enhance network security. IPSec is a good candidate for this purpose. IPSec is a set of protocols developed by the IETF to support secure exchange of packets at the IP layer. It has been deployed widely to implement Virtual Private Networks (VPNs). The design and implementation of IPSec require one to integrate the knowledge of networking, encryption, key management, authentication, and security in OS kernels.

In this lab, students need to implement IPSec in `Minix`, as well as to demonstrate how to use it to set up VPNs. IPSec consists a set of complex protocols; a full implementation is infeasible for a course project. Since the focus of this lab is not on mastering all the details of IPSec, but rather on the integration and application of various security principles, a number of simplifications can be made without compromising the learning objectives.

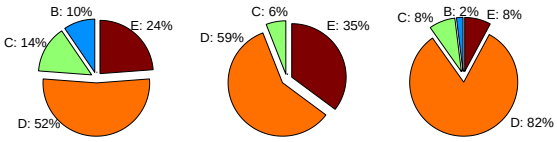
First, IPSec has two types of header (ESP and AH) with two different modes (Tunneling and Transport). Students in this lab only need to implement the ESP header in Tunneling mode. Second, IPSec supports a number of encryption algorithms; in this lab, we only support the AES encryption algorithm. Third, there are many details in IPSec to ensure interoperability among various operating systems. Since interoperability is not a focus of this lab, we make students aware of this issue, but allow them to make reasonable assumption to simplify the interoperability. Fourth, in IPSec, there are two ways for computers to agree upon a shared secret key: one is to use the IKE (Internet Key Exchange) protocol, and the other is through manual configuration. In this lab, students only need to implement the second method, i.e. we assumes that shared secret keys are manually set by system administrators at both ends.

Experience: With such simplifications, most students finished the lab within 5 weeks. Moreover, what interested us



(a) `Set-UID` (b) Capability (c) IPSec

Figure 1: The supporting materials are useful.



(a) `Set-UID` (b) Capability (c) IPSec

Figure 2: The lab is difficult.

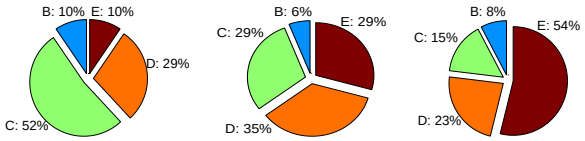
the most was that students were highly motivated. Students attributed their motivation to the fact that this lab was built upon the IP stack: being able to learn the internals of the IP stack has already fascinated many students, much less being able to modify it to enhance security. Most students were proud to put this experience in their resumes, and they told us that recruiters were quite impressed by what they did in this lab. This has reinforced our belief that students get motivated when a security lab is based on a meaningful and useful system, such as TCP/IP, operating systems, etc. Our capability lab and encrypted file system lab have also reinforced such a belief.

3.4 Other Labs

Encrypted File System (EFS): The learning objective of this lab is for students to demonstrate how to integrate encryption with systems to protect confidentiality. EFS is a mechanism to protect confidential files from being compromised when file storages (e.g., hard disks and flash memories) are lost or stolen. EFS builds encryption into file systems, so files are encrypted/decrypted on the fly before they are written to or read from their storages. The process is transparent to users. EFS has been implemented in a number of operating systems, including `Solaris`, `Windows NT`, and `Linux`. In this lab, students need to implement EFS for `Minix`. This lab takes 4-5 weeks to finish.

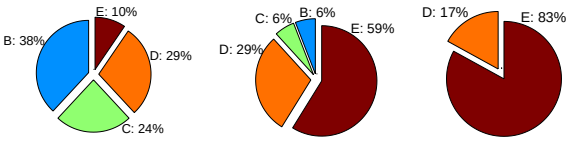
Role-Based Access Control (RBAC): RBAC has become the predominant model for advanced access control, and has been implemented in `Fedora Linux` and `Trusted Solaris`. We have integrated the RBAC concept to our capability lab; it results in a more comprehensive lab.

Sandbox Lab: The learning objective of this lab is for students to substantiate an essential security engineering principle, the *compartmentalization* principle. This principle is illustrated by the *Sandbox* mechanism in computer systems. It is intended to provide a safe place to run untrusted programs. Almost all the `Unix` systems have a simple built-in sandbox mechanism, called `chroot jail`. In this explo-



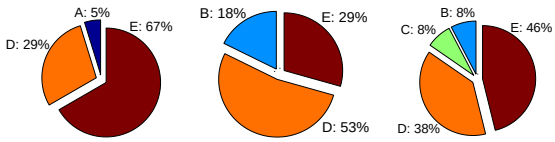
(a) `Set-UID` (b) Capability (c) IPSec

Figure 3: Your level of interest in this lab is high.



(a) `Set-UID` (b) Capability (c) IPSec

Figure 4: The lab is a valuable part of this course.



(a) `Set-UID` (b) Capability (c) IPSec

Figure 5: The lab sparks your interest in computer security.

ration lab, we have designed a series of tasks that are intended to help students understand the implementation of `chroot` and its existing loopholes.

Four Other Labs: We have also developed four other labs, including one for implementing an extended Access Control List for `Minix`, one for exploring Pluggable Authentication Modules (PAM) in `Linux`, one for exploring various vulnerabilities in `Minix`, and another for implementing a simple sandbox mechanism in `Minix`.

4. EVALUATION

We conducted an anonymous survey after students finish each lab. Each survey consists of a number of statements, and students need to choose how much they agree or disagree with the statements. They can choose the following: `(A) Strongly disagree, (B) Disagree, (C) Neutral, (D) Agree, (E) Strongly agree.` The partial results of these surveys for the `Set-UID` lab, capability lab, and IPSec lab are plotted in Figures 1 to 5. The average number of students participating in each lab is 30.

Figure 1 measures whether the supporting materials are satisfactory. From Figure 1(c), we realized that students were not happy with the supporting materials in the IPSec lab. After interviewing students, we found out that, due to the lack of documentation, students spent too much time on figuring out how packets flow within the IP stack in `Minix`. We decided that this part was not essential to the security principles intended in this lab. We plan to develop supporting materials to reduce the time spent on this subject.

Figure 2 measures how difficult each lab is. The data indicates that most students found the labs challenging. However, from their performance, we feel that the labs have successfully pushed students, but without causing them to fail. Most students have succeeded in these labs.

Figure 3 to 5 evaluate the students' perspective in our labs, including how interested they are, whether they think the labs are worthwhile, and whether these labs spark their interests in computer security. From the results, we can see that students' responses are quite positive.

5. CONCLUSION AND FUTURE WORK

We have developed a general laboratory environment for computer security education. Our SEED environment is built upon an instructional OS (`Minix`) and a production OS (`Linux`). The environment can be setup on students' personal computers or public computers with zero software cost. Based on the SEED environment, we have developed ten labs. We tested these labs in our three computer security courses in the last four years; the evaluation results are quite encouraging. Two other universities have started to use the SEED environment and labs in their courses. Several more universities have shown interests in adopting our labs in their courses. In our future work, we plan to further improve the existing labs, as well as developing more labs to cover a broader scope of computer security principles.

6. REFERENCES

[1] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Number 0-521-82060-X. Cambridge University Press, 2nd edition, 2002.

[2] W. A. Christopher, S. J. Procter, and T. E. Anderson. The Nachos instructional operating system. In *Proceedings of the Winter 1993 USENIX Conference*, pages 481–489, San Diego, CA, January, 25-29 1993.

[3] D. Comer. *Operating System Design: the XINU Approach*. Prentice Hall, 1984.

[4] P. J. Denning. Great principles of computing. *Communications of the ACM*, 46(11):15–20, 2003.

[5] J. M. D. Hill, C. A. Carver, Jr., J. W. Humphries, and U. W. Pooch. Using an isolated network laboratory to teach advanced networks and security. In *Proc. of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Charlotte, NC, Feb. 2001.

[6] C. E. Irvine, T. E. Levin, T. D. Nguyen, and G. W. Dinolt. The trusted computing exemplar project. In *Proc. of the 2004 IEEE Systems Man and Cybernetics Information Assurance Workshop*, June 2004.

[7] D. Kolb. *Experiential learning: Experience as the source of learning and development*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[8] W. G. Mitchener and A. Vahdat. A chat room assignment for teaching network security. In *Proc. of the 32nd SIGCSE technical symposium on Computer Science Education*, Charlotte, NC, 2001.

[9] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 2nd edition, 1997.