SEED: A Suite of Instructional Laboratories for Computer Security Education

WENLIANG DU and RONGHUA WANG
Syracuse University

The security and assurance of our computing infrastructure has become a national priority. To address this priority, higher education has gradually incorporated the principles of computer and information security into the mainstream undergraduate and graduate computer science curricula. To achieve effective education, learning security principles must be grounded in experience. This calls for effective laboratory exercises (or course projects). Although a number of laboratories have been designed for security education, they only cover a small portion of the fundamental security principles. Moreover, their underlying lab environments are different, making integration of these laboratories infeasible for a semester-long course. Currently, security laboratories that can be widely adopted are still lacking, and they are in great demand in security education.

We have developed a novel laboratory environment (referred to as SEED). The SEED environment consists of Minix, an instructional operating system (OS), and Linux, a production OS; it takes advantage of the simplicity of Minix and the completeness of Linux, and provides a unified platform to support a rich set of laboratories for computer security education. Based on the SEED environment, we have developed a list of laboratories that cover a wide spectrum of security principles. These labs provide opportunities for students to develop essential skills for secure computing practice. We have been using these labs in our courses during the last five years. This article presents our SEED environment, laboratories, and evaluation results.

Categories and Subject Descriptors: K.3.2 [**Computer & Information Science Education**]: Computer Science Education

General Terms: Security

Additional Key Words and Phrases: security, instructional laboratories, education

ACM Reference Format:
Du, W. and Wang, R. 2008. SEED: A suite of instructional laboratories for computer security education. ACM J. Educ. Resour. Comput. 8, 1, Article 3 (March 2008), 24 pages. DOI = 10.1145/1348713.1348716. http://doi.acm.org/10.1145/1348713.1348716.

The conference version of this paper was published in the *Proceedings of SIGCSE Technical Symposium on Computer Science Education*. March 7-10, 2007, Covington, Kentucky, USA.
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 the Web page http://www.cis.syr.edu/~wedu/seed/.
Contact author: Wenliang Du, 3-114 Sci-Tech Building, Syracuse University, Syracuse, NY 13244; e-mail: wedu@ecs.syr.edu.

1. INTRODUCTION

The importance of experiential learning has long been recognized in the learning theory literature. Dewey (1938) pointed out that learning must be grounded in experience; 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 [Kolb 1984]. 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 [Denning 2003]. Learning from these philosophic views, we should engage students in hands-on experience in security education, and therefore, having effective and well-designed laboratory exercises (or course projects) is critically important to the success of security education [Irvine 1999].

Traditional courses, such as Operating Systems, Compilers, and Network, have effective laboratory exercises that are widely adopted. Unfortunately, computer security education, still at its infancy, has yet had widely-adopted laboratory exercises. Although there does exist a small number of good individual laboratories, 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 environments (operating systems, software, etc.). Usually there is a steep learning curve to learn a new 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 have developed 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 security courses. (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. [Christopher et al. 1993; Comer 1984; Tanenbaum and Woodhull 1997; Howatt 2002]. Our work applies such a pedagogical method in security education. (3) SEED is a low-cost environment. It can be easily set up on students' personal computers with almost zero cost.

Based on the SEED environment, we have developed a list of labs that cover a wide spectrum of principles, innovations, and cutting-edge research ideas essential to computer security. These labs are divided into three classes, each meeting a different educational need. (1) The *design/implementation* labs: The goal of these labs is to achieve learning by system development. They

allow students to apply security principles, concepts, and ideas to build security systems in a lab environment. (2) The *exploration* labs: The goal of these labs is to achieve learning by exploring. They permit students to explore an existing system to understand the intended security principles, concepts, and ideas. Exploration labs are like a "guided tour" of a system, in which students can "touch" and "interact with" the key components of a security system. (3) The *vulnerability* labs: The goal of these labs is to achieve learning from mistakes. Vulnerabilities are often caused by mistakes in design, implementation, and configuration. These labs give students the opportunity to have hands-on experience with real vulnerabilities. In these labs, students need to identify vulnerabilities, develop attacks to exploit vulnerabilities, fix the vulnerabilities, and defend against the attacks.

2. DESIGN PHILOSOPHY AND RELATED WORK

2.1 Design Philosophy

The design of our SEED laboratories is guided by three objectives, which are based upon our firm belief in two teaching philosophies.

> **Philosophy 1:** Computer security education should focus on both the fundamental security principles and security-practice skills. Students should be given opportunities to apply, to integrate, and to experiment with these principles and skills.

Because security education has a much broader scope than many traditional courses, the contents of security courses and the course projects are very diversified. However, regardless of their different "flavor", the underlying principles of security are the same; they are simply taught within different contexts. In our work, we focus on fundamental security principles, including authentication, access control, privilege, auditing, intrusion detection, basic cryptography, security policies, security protocols, vulnerabilities, security engineering principles, security testing, etc. These principles shape the first objective of our work: to develop laboratories that cover a wide spectrum of security principles.

In successful security education, students not only learn how to explain security principles, but more importantly, they learn security practices that characterize our skills as professionals. Peter Denning pointed out in his article, *Great Principles of Computing*, that "our competence is judged not by our ability to explain principles, but by the quality of what we do." He summarized five main categories of computing practice: (1) programming, (2) engineering systems, (3) modeling and validation, (4) innovating, and (5) applying [Denning 2003]. These five skills are the general skills that need to be covered in computer science curricula. In computer security education, not only do we need to foster the development of these general skills, we also want students to attain specialized skills specific to secure-computing practice. We provide a specialized interpretation of the five general skills defined by Denning: (1) Programming: using programming languages securely to build software systems that meet specifications, especially security-related specifications.

(2) Engineering systems: designing and constructing systems and hardware components that are trustworthy. (3) Modeling and validation: formulating security requirements, analyzing security properties, and testing systems for security purposes. (4) Innovating: designing and implementing lasting changes to improve the security of computing systems. (5) Applying: applying security principles to solve real-world security problems. These five skills shape the second objective of our work: to develop laboratories that jointly provide the coverage of these five essential skills in security practice.

> **Philosophy 2:** Computer security education should be integrated into many other courses, including Operating Systems, Networking, Computer Architecture, Compilers, Software Engineering, etc.

Like most other fields, computer security principles can be taught in standalone courses, such as Computer Security and Network Security. However, due to the cross-disciplinary nature of security, these principles should also be taught within the context of other non-security courses, such as Operating Systems, Computer Architecture, Software Engineering, etc. For example, in Software Engineering, students can be exposed to the principles of engineering secure systems; in Computer Architecture, students can ascertain the access control principles through an understanding of how the Intel 80386 security architecture works. Actually, the need for incorporating security across many courses in computer science has already been recognized [Mullins et al. 2002; Vaughn Jr. 2000]; textbooks in many fields have added contents on security in their recent editions. For example, networking textbooks usually have chapters for firewalls and IPSec [Comer 2000]; operating system textbooks generally have chapters for reference monitor, access control, and authentication [Tanenbaum and Woodhull 1997]. However, since security is a relatively new subject in those fields, there are very few effective course projects designed for the security subject within the scope of those fields. Because of this, professors may feel reluctant to incorporate security into their syllabi; even if they do, it will be difficult for them to find suitable course projects.

This philosophy shapes the third objective of our work: the laboratories developed in this project should benefit not only the courses that focus primarily on security, but also those that include security as a component. To achieve this goal, when developing laboratories for various security principles, we produce a variety of laboratories to cover different contexts. For example, for the principle of encryption, we have developed an "Encrypted File System" lab that can be used by operating system courses; we have also developed an "IPSec" lab that can be used by networking courses; needless to say, they can also be used by standalone security courses. To further facilitate the incorporation of security education into many non-security courses, we design each lab as a self-contained module, with its targeted scope and education outcome clearly specified. In this way, instructors can easily identify whether a lab is suitable for their courses.

2.2 Related Work

Many laboratory exercises already exist within security education. An obvious question to ask is whether one can compile a list of existing laboratories for his/her security courses. Unfortunately, doing so is hard, if possible. There are two major reasons.

First, the combined existing labs do not provide comprehensive coverage of the five key skills for secure-computing practice. A popular approach in many security courses is the *attack-based* labs [Hill et al. 2001; Schafer et al. 2001; Micco and Rossman 2002; Hu et al. 2004; Wagner and Wudi 2004; Lie 2005], in which students analyze systems, discover their vulnerabilities, and exploit the vulnerabilities. These labs primarily focus on the "modeling and validation" skills formulated by Denning. The second approach is the *administration-based* labs [George and Valeva 2006; Mayo and Kearns 1999; O'Leary 2006; Crowley 2004; Romney and Stevenson 2004], in which students learn to use security tools to enhance the security of a system. These labs primarily focus on the "application" skills. The third approach is to let students use various security tools, such as firewalls and intrusion detection systems [Ross 2005]; the goal is to achieve learning by playing with a system. The fourth approach is for students to design and implement systems with security components [Joseph et al. 2005; Irvine et al. 2004; Mitchener and Vahdat 2001]. The last two approaches cover system building and secure programming skills, but the choices are quite limited. Moreover, each of these labs tends to cover several security principles; while this makes them more appropriate for comprehensive labs, they are quite inappropriate for focus labs, the goal of which is to target an individual security principle. Our labs fill the void of the existing lab practices. In addition to a variety of comprehensive labs, we have also developed labs for individual security principles.

Second, putting existing labs together for a semester-long course is further complicated by the fact that these labs were developed on a variety of platforms: some use different applications [Mitchener and Vahdat 2001; Irvine and Thompson 2003; Lie 2005; Joseph et al. 2005; Memon 2005; Bishop 1997]; some use different operating systems [Hill et al. 2001; Irvine et al. 2004; Mayo and Kearns 1999; O'Leary 2006; Crowley 2004; Romney and Stevenson 2004; Wagner and Wudi 2004]; some use databases [George and Valeva 2006]. Learning these different platforms is not a trivial task. If students spend too much time learning a new platform for each lab, they will not have sufficient time to work on the tasks that are essential to security. A unified environment would be preferable.

3. THE SEED LAB ENVIRONMENT

3.1 A Unified Platform

To achieve our objectives, we decide to use a unified platform as the basis, and have each lab either deal with an existing component of the platform or

introduce a new component. To provide a comprehensive coverage (i.e., the first objective of our design), a unified platform should have either already covered or provided a fertile ground to cover all the essential security principles. We chose open-source operating systems as our platform, and used the components of operating systems (including the application programs in OSes) as the basis for each lab. This is primarily because an operating system is a single system that provides the most comprehensive coverage of security principles. Many security principles are the basics for all the trustworthy OSes, such as authentication, access control, auditing, encryption, vulnerability, etc. Many other security principles, although not in every OS, exist in some modern OSes, such as capability, mandatory access control, role-based access control, intrusion detection, packet filtering, IPSec, etc. Therefore, operating systems are the best candidate for our unified platform.

In addition to supporting the security principles, a platform should also be effective for students to practice and attain the five computing practice skills described in our second design objective. To foster these skills, based on the platform, we ask students to conduct one or some of the following tasks in each lab: (1) explore (or "play with") an existing security component of the OS; (2) test a security component; (3) identify and exploit vulnerabilities in the OS; (4) modify an existing security component; and (5) design and implement a new security functionality. 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 4, 5, and perhaps 2 (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. This is because most of the security components that students need to address are not standalone 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, but understanding such 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 [Tanenbaum and Woodhull 1997], Nachos [Christopher et al. 1993], and Xinu [Comer 1984]; compiler courses often use instructional languages such as miniJava [Appel and Palsberg 2002]. These systems are designed primarily 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 [Christopher

et al. 1993; Comer 1984; Tanenbaum and Woodhull 1997; Howatt 2002]. 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 [Felder and Silverman 1988].

We use Minix, a widely used instructional operating system in our security education. However, there are two limitations of instructional OS. First, instructional OSes do not have the many interesting security systems that production OSes offer, such as mandatory access control and capability-based system. These systems may provide invaluable experience for students. Second, if we only use instructional OSes, we will not be able to use the many tools developed by the open-source community for the production OSes, while these tools would be quite useful for our labs.

The aforementioned limitations of instructional OSes are inevitable since limiting the features actually defines instructional OSes, and ultimately makes their use a successful pedagogical method. Taking a closer look at the purpose of these instructional OSes, we realized that their strength lies in the capability to support design and implementation tasks, rather than the exploration tasks; the former tasks require a significant amount of effort in coding and code reading, which is not necessary for the latter tasks. For security education, a unified platform should be able to support both design/implementation and playing/exploration types of activity.

3.2 The SEED Environment

The limitations of instructional OSes are the strength of production OSes, and vice versa. To harness their power, we use a hybrid approach to construct our SEED instructional laboratory environment. In the SEED environment, we run both types of OSes. 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. Borrowing the terminology from software engineering, we call this type of lab the *white-box* lab. In the SEED environment, we also use a production OS.[1] The purpose of using this type of OS is two-fold. First, these OSes provide a rich set of security systems. We will 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 systems. We call this type of lab the *black-box* lab. Second, the open-source community has developed many useful tools for production OSes, such as packet analyzers, packet filtering, system monitoring tools, and intrusion detection systems. These tools will be extremely useful for many of our labs. By using production OSes, we can take advantage of these tools. To summarize, we use Minix as the platform for white-box labs, and Linux for black-box labs.

[1]In this project, we mostly use Linux. However, other types of Unix OS, such as FreeBSD can also be used. Actually, some students in our class conducted their labs in several versions of Unix simultaneously.

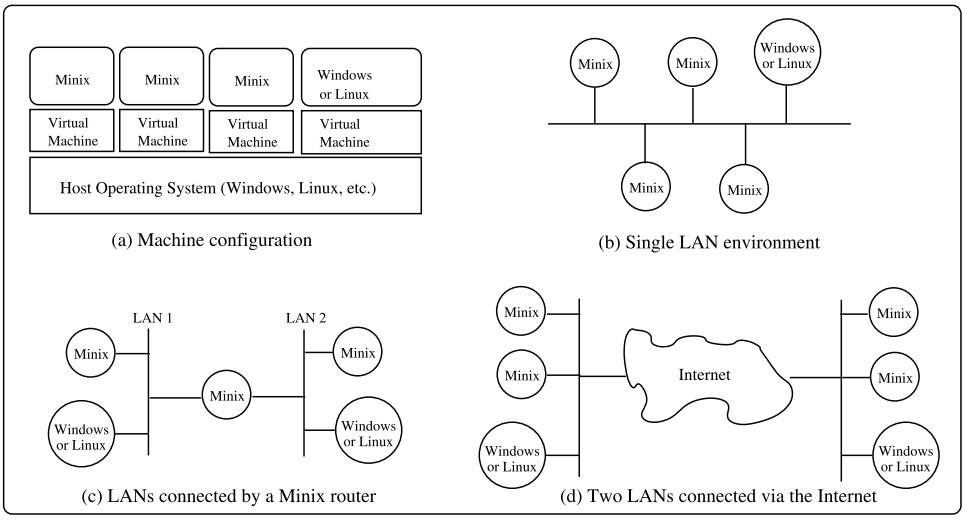


Fig. 1. The SEED environment.

Virtual Machines. To be able to run Minix and Linux (sometimes multiple instances of them) conveniently in a general computing environment, we use 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. These virtual machines and virtual networks form our SEED instructional environment. Figure 1(a) illustrates the virtual environment of SEED. Figure 1(b, c, d) depict a few examples of the virtual network configuration in the seed environment.

The SEED environment can be created using virtual machine software, such as VMware and Virtual PC. The cost of these software is minimal: VMware established an academic program that makes the license of all VMware software free for educational uses, while Virtual PC software can be downloaded free of charge from Microsoft's website. In our SEED environment, we choose VMware as our virtual machine software.

Benefiting from virtual machine technologies, the SEED lab environment does not require a dedicated 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 the rest 15% prefer to use public computers for the labs, but feel okay to use their own computers. An alternative is to install VMware on the machines in public laboratories. However, since students need their own individual virtual machines, and 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. This can be solved with the help of less expensive portable storage media: students can store their

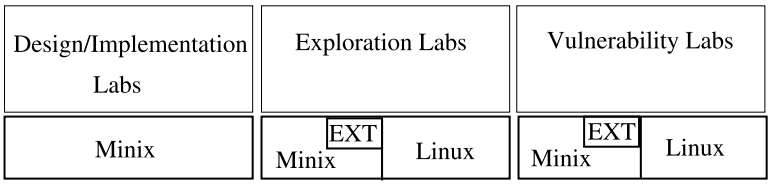


Fig. 2. Three types of labs ("EXT" in the Minix box represents the extensions of the Minix operating system developed in the Design/Implementation labs).

virtual machines on portable hard-disks (cost is less than $100), and work on their lab assignments on any public machines that have VMware installed.

4. THREE TYPES OF LABORATORIES

Based on student attainment of the five essential skills described in our design philosophies, we have developed three types of labs: design/implementation labs, exploration labs, and vulnerability labs. (1) The objective of the design/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 this type of labs include the systems that come with the underlying operating systems (both Minix and Linux) and the systems that we build in the design/implementation labs. (3) The objective of the vulnerability labs is to provide students with first-hand experience on various vulnerabilities, attacks, and countermeasures. Students can discover why some design or implementation errors can lead to vulnerabilities, how vulnerabilities cause security breaches, how they are exploited, and how to apply the security testing principles to detect vulnerabilities. More importantly, students can learn from other people's mistakes. The relationship of these 3 lab types and their underlying platforms are depicted in Figure 2.

4.1 The Design/Implementation Labs

Design/Implementation labs achieve learning by system development. For most of the security principles, we can find some existing systems that exemplify the necessary principles. For example, Mandatory Access Control (MAC) has been implemented in the Security-Enhanced Linux [Loscocco and Smalley 2001], which is now incorporated into Fedora Linux [Fedora Project 2005]; Role-Base Access Control (RBAC) [Ferraiolo and Kuhn 1992] has been implemented in both Fedora Linux and Solaris [SUN Microsystems, Inc. 2001]. To learn these principles, it is desirable for students to have opportunities to actually build these systems (for Minix). However, systems like MAC and RBAC are the results of several years' efforts by a large group; to make the lab tasks

achievable within 4-6 weeks by a small group of students, these systems must be simplified. We have identified a list of systems that cover a broad scope of security principles. We defined a simplified version for each of these systems, and developed the design/implementation labs based on the simplified systems. These labs are desirable for final projects, which usually require four to six weeks to finish.

4.2 The Exploration Labs

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 exploration labs provide a feasible means by which the students have "a direct encounter with the phenomena being studied rather than merely thinking about the encounter, or only considering the possibility of doing something about it" [Borzak 1981]. To use an analogy, exploration labs are like "guided tours"; they guide students through a security system, and on their way, students will be able to "touch" and "interact with" the key components of the system.

The target systems for students to explore include the systems that come with the underlying operating systems (both Minix and Linux). Moreover, all the systems that we built in the design/implementation labs can also be used for exploration labs.

The exploration labs are more desirable for undergraduate students in computer science, non-CS students, and students with weak programming background. In addition, even for students with strong programming backgrounds, instructors may choose this type of labs over the design/implementation labs if they expect to limit the time devoted to a particular targeted security principle. Most exploration labs need one to two weeks.

4.3 The Vulnerability Labs

People learn from mistakes. In security education, we study mistakes that lead to software vulnerabilities. Studying mistakes from the past not only help students understand why systems are vulnerable, why a "seemly-benign" mistake can turn into a disaster, and why many security mechanisms are needed. More importantly, it also helps students learn the common patterns of vulnerabilities, so they can avoid making similar mistakes in the future. Moreover, using vulnerabilities as case studies, students can learn the principles of secure design, secure programming, and security testing.

We have developed vulnerability labs to provide a wide range of vulnerabilities to cover most of the common vulnerability patterns. Using these labs, students can gain first-hand experience on vulnerabilities, as well as on attacks against these vulnerabilities. Each lab targets a specific type of vulnerability. Students are given a system with hidden vulnerabilities. Based upon the hints provided, students need to discover these vulnerabilities; then they need to explore ways to exploit the identified vulnerabilities. Finally, students need to

demonstrate ways to avoid these vulnerabilities in design and programming and to protect systems against these attacks.

We develop all the vulnerability labs within the SEED environment, based on the Minix and Linux operating systems. There are a number of vulnerabilities that still exist in the modern OSes. For example, the TCP RESET attack and the TCP session hijacking attack both target a design weakness of the TCP protocol that cannot be fixed without changing the protocol. Therefore, both attacks are still effective against modern operating systems. For these vulnerabilities, Linux is an ideal platform for our labs. However, many representative vulnerabilities that we would like to use in the vulnerability labs have already been fixed in Linux. We turn to Minix because it is much less frequently updated than the production OSes, and thus it still preserves many vulnerabilities that have been fixed in the production OSes.

Unfortunately, using the existing Minix and Linux alone is still not enough to provide a wide coverage of various vulnerabilities. One solution is to use an old version of Linux that have most of the vulnerabilities. However, most virtual machine software do not support very old OS versions. To be able to conduct vulnerability labs in the SEED environment, we use a *fault injection approach*; namely, based on a selected vulnerability that has occurred in a real-world system, we "port" it to our SEED environment, and thus inject the same vulnerability to Minix or Linux. For example, `at` is a Unix command that allows users to execute commands at a later time. A number of vulnerabilities have occurred in this program, including "race condition" and "environment variables". The `at` program has been patched to fix those vulnerabilities; however, by removing the patches, we can recreate those vulnerabilities, and use the program for our labs.

We have developed 5 vulnerability labs, all of which are carried out in the SEED environment. Some labs use either Minix or Linux, and some use both. For most of the labs, if conducted in a supervised lab environment, they can be finished within a few hours; otherwise, if conducted outside of labs, they can be finished within one week.

5. THE SEED LABORATORIES

During the last five years, we have developed 12 laboratories, most of which have already been used (tested) in our classroom environments, and some have been used for multiple times. We will not describe each of them in full details in this paper; instead, we choose some of our representative labs, and describe them in details. For the other labs, we only provide a brief description. Readers can get the full lab descriptions, activities, guidelines, and supporting materials from our Web site.[2]

5.1 Capability Design/Implementation Lab

The learning objective of this design/implementation lab is for students to apply the capability concept to achieve access control in system.

[2]http://www.cis.syr.edu/~wedu/seed/

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' capabilities, and decides whether to grant the access or not.

5.1.1 *Expectations.* In this lab, students need to implement a simplified capability system for Minix. To make this lab accomplishable within four weeks, we have only defined 5 capabilities. Obviously they do not cover all of the root's privileges; however, it is sufficient for students to learn the principle with these five capabilities: (1) CAP_READ: Allow read permission on all files and directories. (2) CAP_CHOWN: Allow to change file ownership and group ownership. (3) CAP_SETUID: Allow to change the effective user ID. (4) CAP_KILL: Allow killing of any process. (5) CAP_SHUTDOWN: Allow to shutdown the system.

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 disabling, enabling, deleting, delegating, and revoking capabilities.

5.1.2 *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.

5.2 Role-Based Access Control Design/Implementation Lab

The learning objective of this lab is for students to integrate the capability and the Role-Based Access Control (RBAC) mechanisms to enhance system security.

Role-Base Access Control (RBAC), introduced in 1992 by Ferraiolo and Kuhn, can reduce the complexity and cost of security administration in large applications. RBAC has been implemented in Fedora Linux and Trusted Solaris. With RBAC, permissions are not assigned directly to users; instead, they are assigned to *roles*, and roles are assigned to users. Roles greatly simplify the management on permissions.

5.2.1 *Expectations.* In this lab, students need to implement RBAC for Minix. The specific RBAC model is based on the NIST RBAC standard [Ferraiolo et al. 2001]. We define permissions as capabilities. Students first need to implement the five capabilities as defined in the capability lab, not including the management part. In addition, students need to implement 80 dummy capabilities, which have no impact on access control (we "pretend" that they can affect access control). We want to have a significant number of capabilities in this lab to make the management more interesting. With this many capabilities and users, it is difficult to manage the relationship between capabilities and users. The management problem is aggravated in a dynamic system, where users' required privileges can change quite frequently. RBAC provides an effective way to solve this problem. By implementing RBAC in Minix, students will not only learn the RBAC principles, but also encounter various interesting issues when developing an access control system.

The NIST RBAC model is quite complicated, so we only ask students to implement a subset of the model, including the Core RBAC, support for the separation of duty, and role management (e.g., disabling roles, enabling roles, dropping roles, delegating roles, revoking roles, etc.). The lab is intended for four to six weeks.

5.2.2 *Experience.* This is so far the most difficult lab among all our SEED labs. The most challenging part is the design, because a bad design can have potential loopholes. To make things right, students have to deal with a number of issues, including how processes are initialized with roles, where to store roles, how to co-exist with Minix's existing access control mechanism (i.e., Access Control List), how to support the Set-uid mechanism using roles, and so on.

To provide students with more guidance, we asked TA to give four lab sessions, two on design and two on implementation. Students discussed their design and implementation issues in the lab sessions. As results, 80% of the students successfully finished the labs, and another 10% received significant partial credits.

5.3 IPSec Design/Implementation 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.

5.3.1 *Expectations.* 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 of 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 headers (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 assume that shared secret keys are manually set by system administrators at both ends.

5.3.2 *Experience.* With such simplifications, most students finished the lab within 5 weeks. What interested us 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. Several of our other labs have also reinforced the same belief.

Another experience worth mentioning is the virtual machine software. To test their IPSec implementations, students need at least 2 machines; to test their VPN implementations, they need 3-4 machines. Virtual machine software clearly demonstrates its advantage in this situation: students can test all of these using several virtual machines created within a single computer.

5.4 Encrypted File System Design/Implementation Lab

The learning objective of this lab is for students to integrate encryption with systems to protect data.

Encrypted File System (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. In an EFS, files on disks are all encrypted; nobody can

decrypt the files without knowing the required secret. Therefore, even if an EFS disk is stolen, its contents are kept confidential. When legitimate users use the files in EFS, the users do not need to conduct encryption/decryption explicitly. When they read a file (encrypted) using normal editor software, EFS will automatically decrypt the file contents before giving them to the software; similarly, EFS will automatically encrypt the file contents when users write to a file. All these happen on the fly, and must be transparent to users.

5.4.1 *Expectations.* In this lab, students need to implement EFS for Minix. From the lab, students can gain experience on the use of encryption algorithms; they will deal with several issues, such as padding, encryption mode, IV (initial vector), etc. More importantly, students need to think about various key management problems. In particular, they need to decide where to store the keys, how to store the keys, whether to use a different key for each user, whether to use a different key for each file, and how to update keys. Furthermore, students need to decide how users should be authenticated before they can encrypt/decrypt files in EFS. As a bonus, students are encouraged to think about how to support file sharing in EFS; namely, if a file is group readable, how can we allow all the members in the group to decrypt the file. This is quite a challenging problem in EFS design.

5.5 Sandbox Design/Implementation Lab

The learning objective of this lab is for students to substantiate an essential security engineering principle: the *compartmentalization* principle.

The compartmentalization principle is illustrated by the *Sandbox* mechanism in computer systems. It is intended to provide a safe place to run untrusted programs. When we need to run an untrusted program, we would like not to run the program in our own accounts, because the program might be malicious and can compromise the security of our accounts. Instead, it is more desirable if the operating system can create a new user ID for us, and allows us to run the program using this new user ID. Since the new user ID is only temporary (it will disappear after the process ends), the damage caused by the untrusted program is quite limited (e.g., the program cannot read/modify any file unless a file is world-readable/writable). This new user ID is like the special nobody user in most of the Unix operating systems.

5.5.1 *Expectations.* In this lab, students need to implement a simple sandbox for Minix, and they need to design and implement a Set-RandomUID mechanism to achieve the previous mentioned goal. When a Set-RandomUID program is executed, the operating system randomly generates a non-existing user ID, and runs the program with this new user ID as the effective user ID. We can consider Set-RandomUID as an opposite to the Set-uid mechanism: Set-uid allows users to escalate their privileges, while Set-RandomUID allows users to downgrade their privileges. The implementation of Set-RandomUID can be similar to that of Set-uid. However, students need to watch out for potential loopholes. The lab is intended for one to two weeks.

5.6 Set-uid Exploration 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 runs, 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 security breaches.

5.6.1 *Expectations.* This is an exploration lab that takes one to two weeks. 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. First, students need to figure out why some commands, such as `chsh`, `passwd`, and `su` need to be Set-uid programs. Second, using the Minix source code, students need to find out how Set-uid is implemented in the Operating System and how it affects the access control. Third, to demonstrate why Set-uid programs must be very carefully written, students are given a vulnerable program that mistakenly calls `system("ls")` to invoke the `/bin/ls` command (instead of calling `system("/bin/ls")`). Students need to exploit this vulnerability in both Linux and Minix. Fourth, we use another program to demonstrate that it is safer to use `execve()` rather than `system()` when invoking a program within a process. Students will see the different results of these two different calls. Finally, students are required to run a program that permanently relinquishes its root privileges (using the `setuid()` system call). However, due to the inappropriate cleanup, the process still has privileges. Students need to explain their observations.

5.7 Intel 80386 Protection Mode Exploration Lab

The learning objective of this lab is for students to investigate how 80386 protection mode works, why it is designed in the way it is, and how hardware protection is essential for building a TCB (Trusted Computing Base). 80386 protection mode is quite complicated, and without really "touching" it, its essential concepts, such as rings, memory protection, descriptor tables, etc., can be very abstract.

5.7.1 *Expectations.* In this lab, students will interact with the key elements of the Intel 80386 protection mode. The design of this lab is not easy. Since there is no actual need for users to interact with the protection mode, operating systems do not provide an interface for this purpose. To really see how protection mode works, interaction is essential. To achieve this, we poke a hole in Minix, which is basically an interface that allows students to "enter" the protection mode, see how it works, and observe how things can become different when they make some changes. For example, we implement a special

system call, which allows users to change the ring level of their processes. We ask students to report what they can achieve with such a system call. We also provide a mechanism for students to change the contents of GDT (Global Descriptor Table) and LDT (Local Descriptor Table), which are two critical tables in protection mode to achieve memory protection. Students need to find out what they can achieve with such a mechanism.

Other than for computer security courses, this lab has a potential interest to the Computer Architecture instructors, who want to integrate security in their courses.

5.8 Buffer-Overflow Vulnerability Lab

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerabilities, attacks, and countermeasures.

5.8.1 *Expectations.* In this lab, students are given a Set-uid program that has a buffer-overflow vulnerability; they are also given a partially-completed exploit code. They need to complete the exploit code, and use it to successfully launch a root shell by attacking the vulnerable program. To do that, students need to construct an array correctly using the return address that they have to guess and the shell code that is provided. Then they need to use this array to overflow the buffer in the Set-uid program.

Students are also asked to enable the countermeasures that are already in Fedora Linux, including the non-executable stack and the address space randomization. Students will then repeat their attacks, and explain why the attacks now become much more difficult. To show that non-executable stack alone is not a strong countermeasure, we ask students to implement a `return-to-libc` attack on the buffer overflow vulnerability. This attack does not inject shell code onto the stack; instead, it "tricks" the vulnerable program to jump to an `libc` function, which is already loaded in the memory.

5.9 Format-String Vulnerability Lab

The learning objective of this lab is for students to gain the first-hand experience on format-string vulnerability, attacks, and countermeasures.

The format-string vulnerability is caused by code like `printf(user_input)`, where the contents of variable `user_input` is provided by users. When this program is running with privileges (e.g., Set-uid program), this `printf` statement can lead to one of the following: (1) crash the program, (2) read from an arbitrary memory place, and (3) modify the values of in an arbitrary memory place. The exercises in this are designed to help students gain actual experience on such a vulnerability.

5.9.1 *Expectations.* Students are given a vulnerable program (Set-uid) with the format-string vulnerability. In this program, there is an array located on the heap, which holds a number of secrets. The address of this array is stored in a variable located on the stack. The confidentiality and integrity of these secrets are important to the privileged program. By exploiting the format-string vulnerability, students need to cause the program to print out

one secret and then modify another. The main challenge in this lab is to construct an appropriate format string to achieve the above goals. Moreover, similar to the buffer-overflow vulnerability, students will see how address space randomization makes such attack more difficult.

5.10 Race-Condition Vulnerability Lab

The learning objective of this lab is for students to gain the first-hand experience on race-condition vulnerability, attacks, and countermeasures. Race condition is an anomaly in the system whereby the output depends on the sequence or timing of inputs.

5.10.1 *Expectations.* Students are given a Set-uid program with a race-condition vulnerability, and they need to write a script to exploit this vulnerability and accomplish the following tasks: (1) overwrite any file that belongs to root, and then (2) become the root. Unlike many other attacks, to succeed, students need to run their script many times. At the beginning, students are expected to add an idle loop in the vulnerable program between the time of check and the time of use. By increasing the length of the loop time, the chance of success becomes higher. Once students have succeeded, the loop is removed, and students are required to report how long it takes to succeed in their attacks.

5.11 chroot Sandbox Vulnerability Lab

The learning objective of this lab is for students to understand the implementation and potential security problems of a sandbox mechanism.

Almost all the Unix systems have a simple built-in sandbox mechanism, called chroot jail. By redefining the meaning of `"/"`, chroot creates an environment in which the actions of an untrusted process are restricted, and such restriction protects the system from untrusted programs. There are a number of problems with this sandbox implementation, the worst of which allows the prisoned programs to break out of the prison.

5.11.1 *Expectations.* Equipped with the exploit knowledge we covered in our lectures, students need to write a malicious code to implement the exploit in both Minix and Linux. The code will be run in the chroot prison with the root privileges; a successful exploit should be able to break out of that prison.

5.12 TCP/IP Vulnerability Lab

The learning objective of this lab is for students to gain first-hand experience on the vulnerabilities in TCP/IP protocols, as well as on attacks against these vulnerabilities.

5.12.1 *Expectations.* Students are provided with a tool called Netwox, which can be used to construct any arbitrary packets. They need to use this tool to conduct the following attacks on the TCP/IP protocols of both Minix and Linux: (1) SYN flooding attack, (2) TCP RESET attack, (3) TCP session hijacking attack, (4) IP fragmentation attacks, (5) Smurf attacks, and (6) ARP Cache

poisoning attacks. All these attacks are conducted in the SEED environment using virtual machines.

5.13 Labs to Be Developed

To cover a wider spectrum of security principles, as well as to reflect the emerging trends in computer security, we will develop many more labs in our future work. These labs will include more recent vulnerabilities and attacks, such as SQL injection attacks, cross-site scripting attacks, integer overflow attacks, etc. We will also develop more exploration labs to provide more labs for undergraduate security education. For example, we plan to convert all our design/implementation labs to the exploration type, so students (especially undergraduate) do not need to fully implement those systems, but they are still able to learn the concepts and principles by playing with the existing systems.

6. EVALUATION

We started developing SEED laboratories in 2002, mainly for the courses that we taught at Syracuse University. At the beginning, the laboratories were not stabilized, because we had to keep revising them based on the feedbacks from students and the lessons we learned from the actual classroom deployment. Therefore, we only used informal evaluation to collect feedbacks from students; not much attention was put on rigorously evaluating how well each lab achieved its intended objectives. Starting in 2006, as some of our labs became stabilized, we began to conduct formal evaluation of those labs.

6.1 Evaluation Metrics and Methodology

We have identified two factors that are critical for successfully achieving our goal, i.e., to enhance student's learning in security education using the SEED laboratories. These factors include the *efficiency* of the labs and the *effectiveness* of the labs in computer security education.

The efficiency factor focuses on the appropriateness and design of the SEED labs. We quantify the lab efficiency using the following metrics: (1) the level of difficulty of a lab; (2) the usefulness of the supporting materials; (3) the time students spent on a lab; (4) whether the time spent is worthwhile; (5) whether too much time is spent on the tasks that are non-essential to the targeted security principles.

The effectiveness factor assesses student's learning as a result of the project. We quantify the effectiveness using the following metrics: (1) students' level of interest in the lab exercise; (2) students' level of engagement in the lab exercise; (3) level of challenge presented by the lab exercise; (4) amount of effort students exerted in completing the lab exercise; (5) students' level of understanding of the targeted security principles; (6) students' application of the required skills and confidence in their abilities; (7) students' development of related skills: problem solving, critical thinking, creative thinking, ability to think independently; (8) students' perceptions of their learning; (9) students' assessment of the lab as a valuable part of the course.

To evaluate the efficiency and effectiveness of our laboratories, we conducted anonymous surveys after students finished their labs. The survey questionnaires consist of two types of questions: multiple-choice questions and open questions. For multiple-choice questions, we give students a statement, such as "the lab was a valuable part of this course." Students need to choose how much they agree or disagree with the statements.

For open questions, we ask students the followings: (1) Which part of the lab was most time consuming? (2) Which aspects of the lab were most valuable to your learning? (3) What problems did you encounter in completing the lab? (4) What changes could be made to the lab to enhance your learning? (5) What was the most important thing you learned from the lab experience? (6) If you have been interviewed for jobs in the past months, were recruiters particularly interested in this lab?

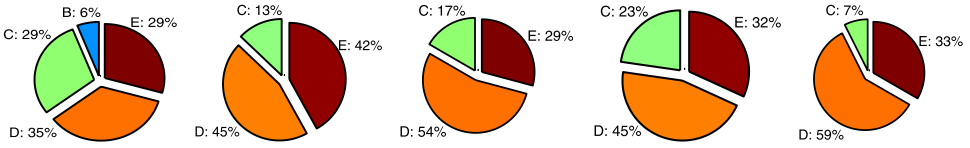
In addition to the surveys, we also rely on the feedbacks from our teaching assistants (TA). Because the TAs can frequently interact with the students, they have the opportunities to identify the common trends and problems among the students. We ask TAs to constantly report their findings to us. This type of evaluation turned out to be very useful, because we could immediately take an action when we saw an emerging problem. Therefore, with TAs' feedbacks, we can correct problems on the fly.

6.2 Efficiency of the Labs

This part of evaluation helps us identify the problems in our lab descriptions, activity designs, guidelines, and supporting materials. Through our evaluations, we were able to identify the bottleneck of the labs, if there is any. We define bottlenecks as the parts of the labs that are time-consuming, necessary, but are not the main objective of the labs, i.e., students cannot get to the main part of the labs without getting over the bottleneck parts first.

We have identified two types of common bottlenecks. The first type is the understanding of the lab requirements. When a lab was first used, a common problem is that the activities were not crystal clear. As a result, students spent a lot of time just figuring out what exactly needed to be done. This problem was mainly caused by the lack of details in our lab descriptions, and we could fix the problem by adding more concrete details in the descriptions.

The other type of bottlenecks is the understanding of the system. This type of bottleneck usually occurs in the design/implementation laboratories. In most of the design and implementation laboratories, students have to build a new security mechanism in Minix. These new security mechanisms are built upon the existing Minix system. Therefore, to accomplish the tasks, students need to figure out how the existing system works. For example, in the IPSec lab, students need to understand how the IP stack works in Minix; in the Encrypted File System lab, students need to understand how the file system works in Minix; in the Capability lab, students need to understand how access control, kernel, and file system works. From our evaluation, we realized that students have spent much more time on these bottlenecks than we expected. In other words, we underestimated the difficult of these bottlenecks.



(a) Capability (18) (b) IPSec (31) (c) `Set-uid` (50) (d) TCP/IP (24) (e) Buffer Overflow (27)

Fig. 3. Students' response to the following statement: "My level of interest in this lab is high."

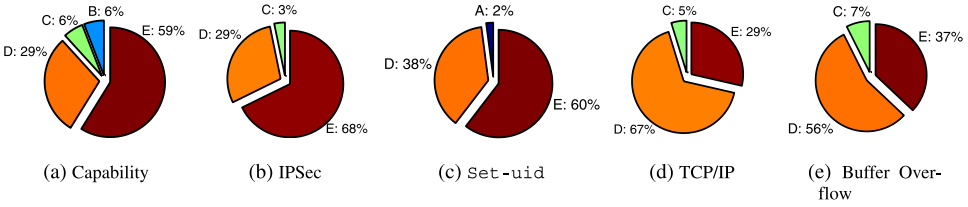
We have developed two strategies to solve our bottleneck problems. If the bottleneck is not essential to security, and removing/simplifying it does not harm the intended objective of the lab, we revise our lab descriptions to either remove the bottleneck or reduce the complexity of the bottleneck. If the bottleneck is really essential to security, we still keep the bottleneck, but we will develop corresponding documents and guidelines, helping students to get over the bottlenecks faster. Namely, to make it possible for students to finish a lab within the intended time frame, we help them to get to the main part of a lab as soon as possible. This way, they can focus on the main principles targeted by the labs. As a result of this strategy, we have developed a number of useful documents during the last five years.

Being able to learn the bottlenecks have helped us tremendously. In 2002, when we started to use SEED labs in our Computer Security class, we could only assign one lab for the entire semester, because students spent too much time on the bottlenecks. As of Spring 2007, students were able to finish five labs in the same course during one semester. Among these five labs, three vulnerability labs were each finished within one week, one exploration in two weeks, and a design/implementation lab in six weeks. Overall, students are doing quite well on these five labs. Comparing to five years ago, this is quite an improvement. The detailed evaluations on the efficiency of each lab are also post in our web pages.

6.3 Effectiveness of the Labs

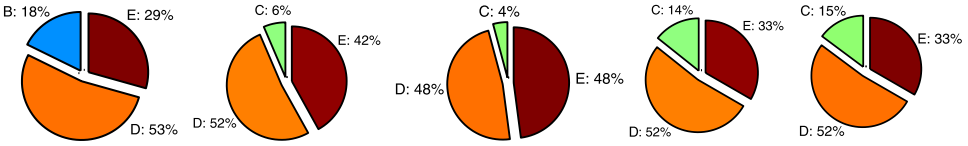
This part of evaluation help us understand how effective our SEED labs are in enhancing students' learning of computer security. We rely mainly on surveys for this part of evaluation. For each survey question, we give students a statement, and they choose how much they agree with the statement by selecting one of the following: (a) Strongly disagree, (b) Disagree, (c) Neutral, (d) Agree, (e) Strongly agree. The statements we used in our survey include the following: (1) My level of interest in this lab is high. (2) The lab was a valuable part of this course. (3) The lab sparks my interest in computer security.

Our survey results for a few selected labs are plotted in Figures 3 to 5. The number of students participating in each survey is listed in the parentheses of the captions in Figure 3. These figures plot the students' perspective in our labs, including how interested they are, whether they think the labs are worthwhile, and whether these labs spark their interest in computer security. From the results, we can see that students' responses are quite positive.



(a) Capability (b) IPSec (c) `Set-uid` (d) TCP/IP (e) Buffer Overflow

Fig. 4. Students' response to the following statement: "The lab was a valuable part of this course."



(a) Capability (b) IPSec (c) `Set-uid` (d) TCP/IP (e) Buffer Over flow

Fig. 5. Students' response to the following statement: "The lab sparks my interest in computer security."

7. 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 12 labs. We tested these labs in our computer security courses in the last five 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 interest in adopting our labs in their courses. In our future work, we plan to further improve the existing labs, as well as develop more labs to cover a broader scope of computer security principles.

REFERENCES

APPEL, A. W. AND PALSBERG, J. 2002. *Modern Compiler Implementation in Java*, 2nd ed. Number 0-521-82060-X. Cambridge University Press. Cambridge, UK.

BISHOP, M. 1997. Computer security in introductory programming classes. In *Proceedings of Workshop on Education in Computer Security (WECS'97)*. Monterey, CA, 1–2.

BORZAK, L. 1981. *Field Study. A Source Book for Experiential Learning*. Beverly Hills: Sage Publications. 9.

CHRISTOPHER, W. A., PROCTER, S. J., AND ANDERSON, T. E. 1993. The Nachos instructional operating system. In *Proceedings of the Winter 1993 USENIX Conference*. San Diego, CA, USA, 481–489. Available at http://http.cs.berkeley.edu/~tea/nachos.

COMER, D. 1984. *Operating System Design: The XINU Approach*. Prentice Hall, Upper Saddle River, NJ.

COMER, D. 2000. *Internetworking With TCP/IP Volume 1: Principles Protocols, and Architecture*, 4th ed. Number 0130183806. Prentice Hall, Upper Saddle River, NJ.

CROWLEY, E. 2004. Experiential learning and security lab design. In *Proceedings of Information Technology Education Annual Conference (SIGITE'04)*. Salt Lake City, Utah, 169–176.

DENNING, P. J. 2003. Great principles of computing. *Comm. ACM 46,* 11 (November), 15–20.

FEDORA PROJECT. 2005. Fedora core 4. Available at http://fedoraproject.org/.

FELDER, R. AND SILVERMAN, L. 1988. Learning and teaching styles in engineering education. *Engin. Educ. 78,* 7, 674–681.

FERRAIOLO, D. AND KUHN, R. 1992. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*. Baltimore, MD, 554–563.

FERRAIOLO, D. F., SANDHU, R., GAVRILA, S., KUHN, D. R., AND CHANDRAMOULI, R. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inform. Syst. Secur. 4,* 3 (August), 224–274.

GEORGE, B. AND VALEVA, A. 2006. A database security course on a shoestring. In *Proceedings of the 37th Technical Symposium on Computer Science Education (SIGCSE'06)*. Houston, Texas.

HILL, J. M. D., JR., C. A. C., HUMPHRIES, J. W., AND POOCH, U. W. 2001. Using an isolated network laboratory to teach advanced networks and security. In *Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE'01)*. Charlotte, NC, 36–40.

HOWATT, J. 2002. Operating systems projects: Minix revisited. *SIGCSE Bulletin–Inroads*, 109–111.

HU, J., MEINEL, C., AND SCHMITT, M. 2004. Tele-lab IT security: an architecture for interactive lessons for security education. In *Proceedings of the 35th Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM Press, Norfolk, Virginia, 412–416.

IRVINE, C. E. 1999. Amplifying security education in the laboratory. In *Proceedings of IFIP TC11 WC11. First World Conference on INFOSEC Education*. Kista, Sweden, 139–146.

IRVINE, C. E., LEVIN, T. E., NGUYEN, T. D., AND DINOLT, G. W. 2004. The trusted computing exemplar project. In *Proceedings of the IEEE Systems Man and Cybernetics Information Assurance Workshop (SMC'04)*. West Point, NY, 109–115.

IRVINE, C. E. AND THOMPSON, M. 2003. Teaching objectives of a simulation game for computer security. In *Proceedings of Informing Science and Information Technology Joint Conference (InSITE'03)*. Pori, Finland.

JOSEPH, A., TYGAR, D., VAZIRANI, U., AND WAGNER, D. CS 194-1, Fall 2005 Computer Security. University of Berkeley. http://www-inst.eecs.berkeley.edu/~cs161/fa05/.

KOLB, D. 1984. *Experiential Learning: Experience as the Source of Learning and Development*. Prentice Hall, Englewood Cliffs, NJ.

LIE, D. 2005. ECE1776: Computer Security, Cryptography and Privacy. University of Toronto. http://www.eecg.toronto.edu/~lie/ECE1776/.

LOSCOCCO, P. AND SMALLEY, S. 2001. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 10th USENIX Conference (FREENIX Track'01)*.

MAYO, J. AND KEARNS, P. 1999. A secure unrestricted advanced systems laboratory. In *Proceedings of the 30th Technical Symposium on Computer Science Education (SIGCSE'99)*. New Orleans, LA, 165–169.

MEMON, N. 2005. CS392/681: Computer Security. http://isis.poly.edu/courses/cs392/.

MICCO, M. AND ROSSMAN, H. 2002. Building a cyberwar lab: lessons learned: teaching cybersecurity principles to undergraduates. In *Proceedings of the 33rd Technical Symposium on Computer Science Education (SIGCSE'02)*. ACM Press, Cincinnati, Kentucky, 23–27.

MITCHENER, W. G. AND VAHDAT, A. 2001. A chat room assignment for teaching network security. In *Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE'01)*. ACM Press, Charlotte, NC, 31–35.

MULLINS, P., WOLFE, J., FRY, M., WYNTERS, E., CALHOUN, W., MONTANTE, R., AND OBLITEY, W. 2002. Panel on integrating security concepts into existing computer courses. In *Proceedings of the 33rd Technical Symposium on Computer Science Education (SIGCSE'02)*. ACM Press, Cincinnati, KY, 365–366.

O'LEARY, M. 2006. A laboratory based capstone course in computer security for undergraduates. In *Proceedings of the 37th Technical Symposium on Computer Science Education (SIGCSE'06)*. Houston, TX.

ROMNEY, G. W. AND STEVENSON, B. R. 2004. An isolated, multi-platform network sandbox for teaching it security system engineers. In *Proceedings of the 5th Conference on Information Technology Education (CITCS'04)*. Salt Lake City, UT.

ROSS, K. 2005. CS393/682: Network Security. http://isis.poly.edu/courses/cs393-s2005/.

SCHAFER, J., RAGSDALE, D. J., SURDU, J. R., AND CARVER, C. A. 2001. The iwar range: a laboratory for undergraduate information assurance education. *J. Comput. Small Coll. 16,* 4, 223–232.

SUN MICROSYSTEMS, INC. 2001. White paper: RBAC in the Solaris operating environment. Available at http://www.sun.com/software/whitepapers/wp-rbac/wp-rbac.pdf.

TANENBAUM, A. S. AND WOODHULL, A. S. 1997. *Operating Systems Design and Implementation*, 2nd ed. Number 0136386776. Prentice Hall, Upper Saddle River, NJ.

VAUGHN JR., R. B. 2000. Application of security to the computing science classroom. In *Proceedings of the 31st Technical Symposium on Computer Science Education (SIGCSE'00)*. Austin, TX, 90–94.

WAGNER, P. J. AND WUDI, J. M. 2004. Designing and implementing a cyberwar laboratory exercise for a computer security course. In *Proceedings of the 35th Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM Press, Norfolk, VA, 402–406.

Received June 2007; revised November 2007; accepted January 2008