Using An Instructional Operating System In Teaching Computer Security Courses *

Wenliang Du
Systems Assurance Institute
Department of Electrical Engineering and Computer Science
Syracuse University
121 Link Hall, Syracuse, NY 13244
Tel: 315-443-9180 Fax: 315-443-1122
Email: wedu@ecs.syr.edu

Abstract

To address national needs for computer security education, many universities have incorporated computer and security courses into their undergraduate and graduate curricula. In these courses, students learn how to design, implement, analyze, test, and operate a system or a network to achieve security. Pedagogical research has shown that effective laboratory exercises are critically important to the success of this type of courses. However, such effective laboratories do not exist in computer security education.

To fill this gap, we have developed an instructional operating system `SMinix` based on the instructional OS (`Minix`) developed for operating system and network courses. We use `SMinix` as a basic framework, and build laboratory exercises on top of the framework; each exercise requires students to add a different security mechanism into the system. The instructional system is designed in such a way that makes it easy for students to focus on the part of the system that are related to the security concepts covered in our classes. The similar approach has proved to be effective in teaching operating system and network courses, but it has not yet been used in teaching computer security courses.

We have already used `SMinix` for our Computer Security course, students have demonstrated great interests in our laboratory exercises.

1 Introduction

The high priority that information security education warrants has been recognized since the early 1990's. The 1991 National Research Council systems security study recommended that "Computer system security and trustworthiness must become higher priorities for ...educators..." [4]. In 1997, Matt Bishop of University of California at Davis noted that few computer science students are required to develop robust, thoroughly tested code, and that until this gap is addressed "security problems will continue to plague computer systems" [1]. In 2001, Gene Spafford of Purdue University's Center for Education and Research in Information Assurance and Security, testified before Congress that "to ensure safe computing, the security (and other desirable properties) must be designed in from the start. To do that, we need to be sure all of our students understand the many concerns of security, privacy, integrity, and reliability" [14].

*The project is supported by Grant DUE-0231122 from the National Science Foundation and by fundings from CASE center.

To address these needs, many universities have incorporated computer and information security courses into their undergraduate and graduate curricula; in many, computer security and network security are the two core courses. These courses teach students how to design, implement, analyze, test, and operate a system or a network with the goal of making them secure. For this type of course, pedagogical research has shown that students' learning is enhanced if they can engage in a significant amount of hands-on exercises. Therefore, effective laboratory exercises (or course projects) are critically important to the success of our computer security education.

Traditional courses, such as operating systems or complier, have effective laboratory exercises, the result of twenty years maturation. In contrast, laboratory designs in security education courses are still embryonic. A variety of approaches are currently used; three of the most frequently used designs are described. (1) The *free-style* approach where instructors allow students to pick whatever security-related topics they like for the course project. (2) The *dedicated computing environment* where students conduct security implementation, analysis and testing [7, 10] in a contained environment. (3) The *build-it-from-scratch* approach, where students are required to build a secure system from scratch [12].

Free-style design projects are effective for creative students who enjoy the freedom if affords; however, most students become frustrated with this strategy because of the difficulty in finding an interesting topic. With the dedicated-environment approach, projects can be very interesting, but the logistical burdens of the laboratory - obtaining, setting up, and managing the computing environment - are considerable for instructors. In addition, course size is constrained by the size of the dedicated environment. The third design approach requires that students spend considerable time on activities irrelevant to security issues, but essential for a meaningful system.

Understanding the drawbacks of the existing approaches, we propose to develop a new approach for the laboratories of computer security courses. We want our labs to have the following properties: (1) It should be well designed, and each laboratory should clearly lay out its goals and tasks. (2) The lab does not need special computing environment or super-user privilege. Students with normal user accounts should be able to carry out the lab assignments. (3) Most importantly, the lab provides an infrastructure to students, so they do not need to implement everything from scratch. Their implementation should be part of a much more complex system, they can immediately see how their implementation behaves without having to build the rest of the complex system. For example, if the task is to build an access control mechanism, after students finish their implementation, they can immediately see how their access control mechanism affects file access within a file system without having to implement the file system.

Such a laboratory design does not exist in computer security education, but similar laboratory designs exist in other more traditional and mature courses, such as operating systems (OS), compilers, and networks. In OS courses, a widely adopted successful practice is to use an instructional OS (e.g. `Minix` [15], `Nachos` [3], and `Xinu` [5]) as the framework and ask students to write significant portions of each major piece of a modern OS. The compiler and network courses adopted a similar approach. Inspired by the success of the instructional OS strategy, we will adapt it to our computer security courses. Specifically, we will provide students with a system as the framework, and then ask them to implement significant portions of each fundamental *security-relevant functionalities* for a system. Although there are a number of instructional systems for OS courses, to our knowledge, this approach has not yet been applied in computer and information security courses.

We identified `Minix` instructional operating system as our basis for two reasons: first, `Minix` is very complete comparing to other unix-style instructional OS; second, `Minix` can run in `Solaris` operating system as a normal process, hence needs no special privilege. Based on `Minix` system, we build our own system by introducing more security mechanisms; we call our system the Security-enhanced `Minix` (`SMinix`). Our design fills what we see as a gap in computer security education:

the lack of effective and efficient laboratory exercises. `SMinix` encompass the fundamental security concepts, principles, and technologies; therefore, it can serve as the framework and platform for students to practice their *security design, implementation, analysis, testing and operation* skills.

`SMinix` provides the framework for fundamental security functionalities, while leaving the detailed implementation to students. For example, while teaching the Discretionary Access Control (DAC) concept, we will give students a version of the `SMinix` system without the DAC mechanism. Students will need to write programs to implement MAC into the system. However, since the basic framework for DAC exists in `SMinix`, students will not be starting from scratch, rather they will build upon the existing framework. Such a framework makes it possible for students to do more exercises.

2 System Design

Using `Minix` directly for students' projects proved to be difficult for the following reasons: (1) Some projects require students to make significant architectural change to the `Minix` system. Although this is also a good practice for students, it might be too difficult. We want to be able to adjust the degree of difficulties so we can customize the projects for students at different levels, e.g. undergraduate level and graduate level. We can also adjust the difficulties based on the amount of time we plan to give to students. (2) Some projects need to modify kernel data structure. For example, to implement a full Access Control List (ACL) mechanism, we need to store the ACL information in the `i-node`; however, the `i-node` data structure in `Minix` does not have a field for that. Adding another entry to this critical data structure is not a trivial job because many other modules of the `Minix` depend on it. Moreover,, modifying kernel data structure is not the emphasis of this course. Therefore, it will better if the system we provide to students already have some *unused* entries in the `i-node` data structure.

For these reasons, we decided to modify `Minix`, especially modify its security architecture and data structures. Our goal is to make the projects easy to be customized, and reduce students' burden on tasks that are not the focus of this class. We call our modified system the `SMinix` operating system.

The `SMinix` system comprises an operating system (OS), with the basic operating system components, the basic security architecture and components, and a number of plug-in modules. The basic OS components all come directly from `Minix`. The security architecture is designed in such a way that all of the projects described in Section 3 can be supported. Each of the plug-in modules implements a specific security mechanism, such as the Access Control, Capabilities, and Sandboxing. The basic `SMinix` is functional with and without the plug-in modules. The overall architecture of the `SMinix` is depicted in Figure 1.

The `SMinix` system is especially useful in teaching students two fundamental skills: (1) security analysis and testing, and (2) security design and implementation.

2.1 Security Analysis and Testing

To master the security analysis and testing skills the students have learned from the class, they need to practice those skills in some system. One way to do this is to give them a system, such as Windows 2000 or Linux and ask them to find security flaws in those systems. Although these systems definitely contain a lot of vulnerabilities, finding them is not a trivial work for those who have just learned the basic skills. Furthermore it is difficult to choose the type of vulnerabilities to make the exercises focus on what students have just learned.

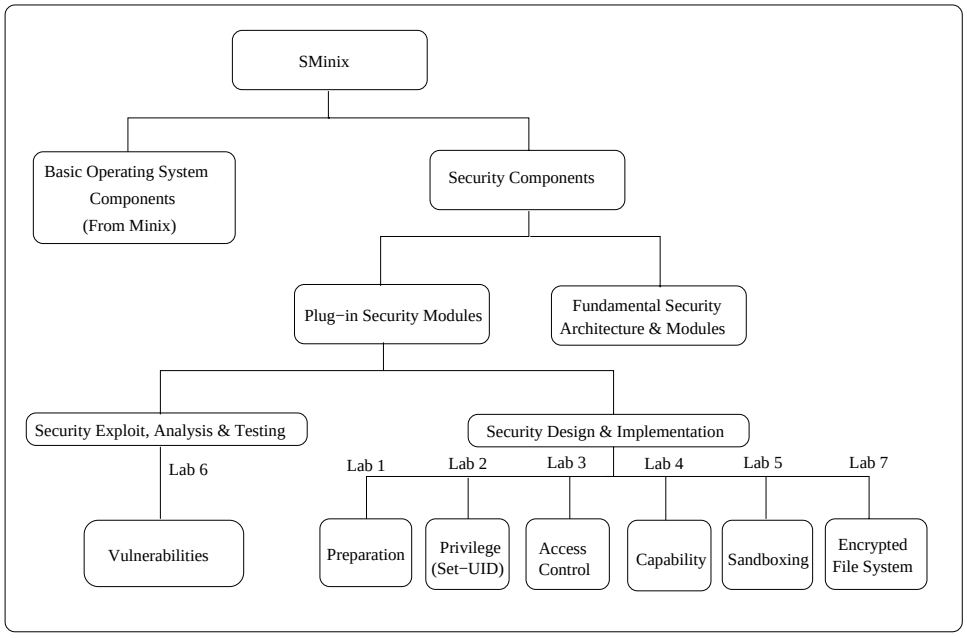

Figure 1: `SMinix` System Architecture

In this project, we will create various versions of the `SMinix` system, each of which contains injected vulnerabilities. Based on the real-life vulnerabilities appearing in many systems, we will modify some components of `SMinix` system to make them contain some specific vulnerabilities. The system is then given to students, who need to find those vulnerabilities and exploit them to compromise the security of the system. The types of vulnerabilities we plan to inject into the `SMinix` system include buffer-overflow errors, race condition errors, sym-link errors, input validation errors, authentication errors, domain errors, and design errors [8]. Before starting these exercises, the students are equipped with theoretical knowledge of these vulnerabilities, the methods of detection and exploitation, and the methodologies of penetration testing and standard security testing. The analysis and testing skills they learned will be used throughout the semester when they test their own implementation in other laboratories.

2.2 Security Design and Implementation

In addition to the security analysis and testing skills, we want to let our students learn how to build a security system, so their systems will not have as many security flaws as what they saw in our previous lab exercises.

A modern operating system usually employs many security mechanism, such as *Authentication, Access Control, Capability, Sandboxing, Secure file system, and Privilege*, all of which will be covered in our course. Ideally, we want to let students implement most of them during one semester period. To learn how long it takes an average student to implement just one of them, we did an experiment in spring 2002 while we taught the computer security course. We asked students to add one of the security mechanism mentioned above to the `Minix` operating system from scratch. The results shows that the amount of coding is not so significant. However, the students spent most of their time figuring out how the new security components that they are suppose to implement fit into the system, where the new components should be put, and how these components interact with the rest of the system. These tasks are very time consuming, they require students to understand

the Minix operating system kernels and its security architectures. We want students to learn all of these, and at the same time, we want them to have opportunities to implement a spectrum of the security mechanisms.

To this end, we decided to teach students the necessary knowledge about the system, lay out the foundation for each security mechanism, and provide enough details about the project, so the students do not need to read through thousands lines of code to figure them out.

To make our goal feasible, we develop the SMinix system in the following way: In the first stage, we will design the SMinix system, so it can contain all of the security mechanisms we mentioned above. This involves the designing of the security architecture and adding the necessary data structure. Each security mechanisms should be implemented as an individual module, and the interaction among modules should be reduced to minimum. In the second stage, we will implement the SMinix system, as well as implementing all of the security mechanisms. We will call each module a complete module. In the third stage, we will turn each complete module into a skeleton module by taking out all of the contents of each complete module and leaving only the interface untouched. We call the resulting system SMinix/S (S means the skeleton). The SMinix/S is still functional, but since the contents of each module is taken out, the security mechanisms will have no effect. Sometimes, we can just take partial contents out of the complete module, and construct a different version of a skeleton module; this version contains more information, and will make lab exercises easier. In another words, we can control the degree of the information that is to be taken out of a complete module for different level of students (graduate and undergraduate).

Based on SMinix/S, we can now develop lab exercises. For each lab exercises, students will need to implement a security mechanism for SMinix/S. Because SMinix/S has already provided a skeleton module for such security mechanism, the students will easily understand the relationship of this module with the rest of the system, and they can quickly start their coding. The coding is just like filling the blank once the students know where to put their codes. This approach dramatically reduces the time for each project, and at the same time, with the help of the skeleton modules, students still learn the important knowledge, such as how the new security modules interact with the whole system.

There are a number of advantages using our approach: (1) SMinix provides students with a structured framework upon which they can build various security mechanisms. (2) The SMinix/S is functional even if the students have not implemented the security modules completely. This gives students quick feedback as to how their implementations work and whether the modules are implemented correctly. (3) Because our design is highly modularized, the instructors can freely plug in any available modules to construct various versions of SMinix systems, each of which has different security functionalities.

3 The SMinix Laboratories

3.1 Laboratories Overview

We have developed a set of laboratory exercises for SMinix system; through these laboratories students will develop a thorough understanding of computer security concepts, principles and techniques by "learning by doing". In particular, we want students to learn various computer vulnerabilities, how those vulnerabilities could be exploited, how to detect them by using various testing methodologies, and how various security mechanisms work.

"Learning by doing" forces students to digest the information presented in class to the point where they can instruct the computer how to apply it. Active learning such as this has a higher

chance of having a lasting effect on students than if the students passively listen to lectures without reinforcement [11].

3.2 Laboratories Setup

All of the laboratories exercises will be conducted in SUN `Solaris` environment using `C` language. Except for giving students more disk space (100 Megabytes) to store the files of `SMinix` system, `SMinix` poses no special requirements on the general `Solaris` computing environment. Therefore, the laboratories can be conducted in the existing computing infrastructure provided by the Department of Electrical Engineering and Computer Science in our school. Such a computing environment is quite general and is available in many universities.

3.3 Laboratories Descriptions

Below is a list of the seven assignments closely related to computer security. The first assignment is intended to familiarize students with the process used in all the laboratories. The sixth lab teach about vulnerabilities and their exploitation. The rest of labs teach various techniques and concepts that are used to enhance system security. Depending on how much time the instructors want the students to spend, they can choose a subset of these seven labs. Instructors can always extend this set of laboratories to include their own laboratories.

Lab 1: Preparation (3 Weeks)

Objectives: We use this project to let students become familiar with the `Minix` operating system. After finishing this project, they should be able to conduct the following tasks in `Minix` operating system: installation, compiling source codes, administration (e.g. adding/removing users), modifying source files, modifying existing system calls, adding new system calls, and understanding the data structure of `i-node` and `process table`. Students with operating system background (the prerequisite for this course) should be able achieve the above goals within two weeks.

Project Description: Students need to finish the following tasks: (1) Compile and install `Minix`, then add three users account to your system. (2) Change the password verification procedure, such that a user is blocked for 15 minutes after three failed trials. (3) Implement system calls to enable users to print out attributes in `i-node` and `process table`. Appropriate security checking should be implemented to make sure a user cannot read other users' information.

Lab 2: `Set-UID` Programs (2 Weeks)

Objectives: `Set-UID` is an important security feature in `Unix` operating system; it is also a good example to show students how privileges could be managed in a system, and what problems a system could have if privileges are not handled properly. We use this project to let students become familiar with the `Set-UID` concept, its implementation, and potential problems.

Project Description: Students need to finish the following tasks: (1) Figure out why `passwd`, `chsh`, `su` commands need to be `Set-UID` programs. What will happen if they are not. (2) Students are given a binary code for `passwd` program, which contains a number of security flaws injected by the instructor. Students need to identify those flaws, and exploit them to gain root privilege. (3) Read the OS source codes of `SMinix`, and figure out how `Set-UID` is implemented in the system. (4) Modify the OS source code to disable the `Set-UID` mechanism.

Lab 3: Access Control (3 Weeks)

Objectives: Access control is an important security mechanism implemented in many systems, and there are different types of access control: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). The goal of this project is two-fold: (1) to get first-hand experience with DAC and MAC, and (2) to be able to implement DAC and MAC.

Project Description: Students are given a version of `SMinix` with no access control mechanism. They need to implement all (or some) of the following access control mechanisms: (1) Abbreviated ACL: the access control is based on three classes: owner, group, and others. (2) Full ACL: the access control can be based on individual users. (3) MAC: design and implement a simple MAC access control mechanism for `SMinix`.

Lab 4: Capability (3 Weeks)

Objectives: The goal of this project is to let students be familiar with the capability concept. We plan to achieve this by asking them to implement a simplified capability mechanism.

Project Description: Students are supposed to implement a simple capability mechanism in `SMinix`. Such a mechanism should support the following: (1) Permission Granting based on capability: To simplify the lab, we fix the permission to a small set. e.g. here are some examples: the capability to execute a specific system call, such as `chown` (change owner), the capability to run `setuid` program, etc. (2) Capability Copying: A process should be able to copy its capability to another process. (3) Capability Amplifying/Reduction A process should be able to amplify or reduce its current capability. For example, a process can temporarily remove its own setuid capability, but later can add it back. Of course, a process cannot add a new capability to itself if it does not already own the capability. (4) Capability Revocation: The root should be able to revoke the capability from current and future processes.

Lab 5: Sandboxing (4 Weeks)

Objectives: A sandbox is an environment in which the actions of a process are restricted according to a security policy [2]. Sandboxing is an important concepts for computer security, we want students to understand such a concept by studying an existing sandbox system and by implementing a sandbox for `SMinix`. In addition to the sandboxing concept, this project also includes a number of other concepts related to security, such as access control, system calls, isolation, and security policies. Therefore, we view this lab as a comprehensive project.

Project Description: Janus [6] implements a sandbox. It is an execution environment in which system calls are trapped and checked. The system is well documented. The task of this lab is to study how Janus work, and then implement a simplified version of Janus for `SMinix`.

Lab 6: Vulnerability Analysis (3 Weeks)

Objectives: The first goal of this lab is to let students gain first-hand experience on software vulnerabilities, to be familiar with a list of common security flaws, to understand how a seemly-not-so-harmful flaw in a program can become a great risk to the system. The second goal of this lab is to give students opportunities to practice their vulnerability analysis and testing skills. Students learn a number of of methodologies from the class, such as vulnerability hypothesis [9], penetration testing methodology [13], code inspection technique, and blackbox and whitebox testing. They need to practice using these methodologies in this lab.

Project Description: The students are given a version of the `SMinix` operating system that is full of injected vulnerabilities. These vulnerabilities simulate system flaws caused by incorrect design, implementation, and configuration. The students are also given some hints, such as a list of possible vulnerabilities, the possible locations of the vulnerable programs, etc. Their task is to find those vulnerabilities.

Lab 7: Encrypted File System (EFS) (4 Weeks)

Objectives: Traditional file system does not encrypt the files that are stored on a disk, so if the disk is stolen, contents of those files can be recovered. An EFS solves this problem by encrypting all files on the disk, such that only users who knows the encryption keys can access the files. The encryption/decryption operations should be transparent to users. Designing and implementing such a system require students to combine together the knowledge about encryption, key management, authentication, access control, security in the OS kernel, and file systems. Therefore this project is meant to be a comprehensive project. We suggest giving this project as a final project after most of the relevant concepts have already been covered in class.

Project Description: Students are given the description of how EFS should work, and their task is to design and implement an EFS for `SMinix` operating system. Students should use their creativity to design the system the way they want.

4 Experience

We did an teaching experiment in the 2002 spring semester when we taught the computer security course (CSE 785). At that time, we have not developed the `SMinix`, so we decided to use the original `Minix` operating system and asked students to modify its kernel to add certain specific security mechanisms into the system. We only give them one project for the whole semester because modifying an OS seems to be a daunting job for most of the students. The students liked the projects very much and were highly motivated. At the end of the semester, the students provided a number of useful suggestions. For example, many students noted, "most of our time was spent on figuring out how such an operating systems work, if somebody or some documentation can explain that to us, we could have done four or five different projects of this type instead of doing one during the whole semester". This observation shapes the goal of our design: we want students to implement a project within two to four weeks using our proposed instructional environment; without which they can only implement one or two projects of during the whole semester.

When we teach teach CSE 785 again this semester (Spring 2003), we are more prepared. We provide students with sufficient information about how `Minix` works, and we even add a lecture to talk about `Minix`. As results, students have gotten familiar with `Minix` within the first three weeks, and are ready for the projects we have designed for them. The same degree of familiarity took my previous students half of a semester due to the lack of information. Moreover, we were able to assign four projects in one semester, including a complicated one, the encrypted file system project. Here are some of lessons we have learned during the last two years:

1. Preparation: The preparation part (Lab 1) is extremely important. If students fails this part, they will spend enormously more time on the subsequent projects. This is very clear when we compare the performance of the students in this year's class with that of the students in 2002. We plan to integrate the materials related to Lab 1 into the lecture, so students can be prepared better.

2. Background Knowledge: We also realized that some students in the class does not know the basic of unix operating system because they have been using Windows most of the time. This brings some challenge because these students do not know how to set up the `PATH` environment variable, how to search for a file, etc. We plan to develop materials to help students get over this first obstacle.

3. Different Level of Difficulties: Because students come from a variety of background, some of them found the last project, the encrypted file system (EFS), extremely difficult, and some cannot finish it. In the future, we plan to design a multi-level requirement for projects that are complicated, such that students with different backgrounds can find a level of requirement that they can achieve. For example, for the EFS project, after realizing this problem, we told some of the students that they can implement their projects without satisfying the "user transparency" requirement; of course, they will receive lower grades than those whose implementations satisfy the requirement.

4. Grading: grading is a challenge if the size of the class is large. The best way for the grading is to ask students to perform a demonstration. However, this is not feasible for a big class. Currently, we ask students to submit their report for each lab, and we selected a few of them for the demonstration if their reports are not convincing. We plan to develop some script to automate the grading.

5 Conclusion and Future Work

We have described the design of `SMinix`, an `Minix` based instructional operating system for teaching computer security courses. We have also described a series of lab assignments based on `SMinix`. Although `SMinix` is not full implemented yet, the experience we obtained by using a prototype is very encouraging, and students in our class have shown great interests in the course.

Currently, we are conducting the full implementation of `SMinix`. The first version will be ready for testing by this September. We will try various means to disseminate our results so other universities can benefit from this NSF funded project. We will also conduct a full evaluation of the effectiveness of using `SMinix` in computer security courses.

References

[1] M. Bishop. Computer security in introductory programming classes. In *Workshop on Education in Computer Security*, pages 1–2, Monterey, CA, USA, January 1997.

[2] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2002.

[3] 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, USA, January, 25-29 1993. Available at `http://http.cs.berkeley.edu/~tea/nachos`.

[4] D. D. Clark. Computers at risk: Safe computing in the informatin age. Washington, DC: National Academy Press, 1991.

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

[6] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, 1996.

[7] 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 *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 36–40, Charlotte, NC, USA, February 2001.

[8] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[9] R. R. Linde. Operating system penetration. In *AFIPS National Computer Conference*, pages pp. 361–368, 1975.

[10] J. Mayo and P. Kearns. A secure unrestricted advanced systems laboratory. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, pages 165–169, New Orleans, USA, March 24-28 1999.

[11] C. Meyers and T. B. Jones. *Promoting Active Learning: Strategies for the College Classroom*. Jossey-Bass, San Francisco, CA, 1993.

[12] W. G. Mitchener and A. Vahdat. A chat room assignment for teaching network security. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 31–35, Charlotte, NC, USA, February 2001.

[13] C. Pfleeger, S. Pfleeger, and M. Theofanos. A methodology for penetration testing. *Computers and Security*, 8(7):613–620, 1989.

[14] E. H. Spafford. February 1997 testimony before the united states house of representatives' subcommittee on technology, computer and network security. Available at http://www.house.gov/science/hearing.htm, 2000.

[15] A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 2nd edition, 1996.