A Novel Approach for Computer Security Education Using `Minix` Instructional Operating System *

Wenliang Du†, Mingdong Shang, and Haizhi Xu
Department of Electrical Engineering and Computer Science,
3-114 Center for Science and Technology,
Syracuse University, Syracuse, NY 13244
Tel: 315-443-9180 Fax: 315-443-1122
Email: {wedu,mshang,hxu02}@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 these types of courses. However, such effective laboratories do not exist in computer security education.

Intrigued by the successful practice in operating system and network courses education, we adopted a similar practice, i.e., building our laboratories based on an instructional operating system. We use `Minix` *operating system as the lab basis, and in each lab we require students to add a different security mechanism to the system. Benefited from the instructional operating system, we design our lab exercises in a way such that students can focus on one or a few specific security concepts while doing each exercise. 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.*

Keywords: Computer security, education, courseware, laboratory projects, and `Minix`.

1 Introduction

The high priority that information security education warrants has been recognized since early 1990's. In 2001, Eugene Spafford, director of the Center for Education and Research in Information Assurance and Security (CERIAS) at Purdue University, 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" [1].

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

†The contact author. Email: wedu@ecs.syr.edu.

To address these needs, many universities have incorporated computer and information security courses into their undergraduate and graduate curricula. In many curricula, computer security and network security are 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 it secure. 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 computer security education.

Traditional courses, such as operating systems, compilers, and networking, have effective laboratory exercises, as 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 the followings: (1) the *free-style* approach, i.e., instructors allow students to pick any security-related topic they are interested in for the course projects; (2) the *dedicated-computing-environment* approach, i.e., students conduct security implementation, analysis and testing [2, 3] in a contained environment; (3) the *build-it-from-scratch* approach, i.e., students build a secure system from scratch [4].

Free-style design projects are effective for creative students; 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, with the logistical burdens of the laboratory–obtaining, setting up, and managing the computing environment. In addition, course size is constrained by the size of the dedicated environment. The third design approach requires students to spend considerable amount of time on activities that are irrelevant to computer security education but are essential for a meaningful and functional system.

The lack of an effective and efficient laboratory for security courses motivated us to consider practices adopted by the traditional mature courses, e.g., operating systems (OS) and compilers. In OS courses, a widely adopted successful practice is using an instructional OS (e.g. `MINIX` [5], `NACHOS` [6], and `XINU` [7]) as a 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 adapt it to our computer security courses. Specifically, we provide students with a system as the framework, and then ask them to implement significant portions of each fundamental *security-relevant functionality* for a system. Although there are a number of instructional systems for OS courses, to our knowledge, this approach has not yet been applied to computer and information security courses.

Our goal is to develop a courseware system, serving as an experimental platform and framework for computer security courses. The courseware is not designed to create new security mechanisms, but to let students practice existing security work. The courseware contains a set of well defined and documented projects for helping students focus on (1) grasping security concepts, principles and technologies; (2) practicing design and implementation of security mechanisms and policies; and (3) analyzing and testing a system for its security properties.

We chose `Minix` as our base system, and have designed a number of laboratory assignments on it. These assignments cover topics ranging from the design and implementation of security mechanisms to the analysis and testing of a system for security purpose. Each assignment can be considered as adding/modifying security mechanisms to `Minix`. To finish each task, students just need to focus on those security mechanisms, with minimum effort on other parts of the system. For example, while learning discretionary access control (DAC), we give students a file system without DAC mechanisms; students only need to design and implement DAC for this existing file system. Students can immediately see how their DAC implementation affect the system. This strategy helps students to stay focus on security concepts.

Our course projects consists of two parts. One part focuses on design and implementation. This part of projects requires students to add new security mechanisms to the underlying `Minix` system to enhance its security. The security mechanisms students need to implement include access control, capability, sandbox, and encrypted file systems. In the second part of our projects, we gave students a modified `Minix` system that contains a number of injected vulnerabilities. Students need to use their skills learned from the lectures to identify, exploit, and fix those vulnerabilities.

Our approach is open-ended, i.e., we can add more laboratory projects to this framework without affecting others. The projects presented in this paper are the result of three years' maturation, with more components added in each year. We are also planning to design a number of network security projects for `Minix` based on the `Minix`'s existing networking functionality.

The paper is organized as the following: Section 2 briefly describes our computer security course. Section 3 describes the design of our courseware. Section 4 describes each of our laboratory projects. Section 5 presents the experiences and lessons we have gained during our three-year practice. Finally, Section 6 concludes the paper and describes the future work.

2 The Computer Security Course

2.1 Scope of the course

Our department offers two graduate courses in security: one is computer security, and the other is network security. The computer security course focuses on the concepts, principles, and techniques for system security, such as *encryption algorithms, authentication, access control, privilege, vulnerabilities, system protection,* etc. Currently, our proposed approach only targets at the computer security course, but we plan to extend this approach to the network security course in our future work.

2.2 Pedagogical Approach

Lecturing on theories, principles and techniques of computer security is not enough for students to understand system security. Students must be able to put what they have learned into use. We use the"learning by doing" approach. It was shown in other studies that this type of active learning approach has a higher chance of having a lasting effect on students than letting students passively listen to lectures without reinforcement [8].

More specifically, we try to use the `Minix` OS as our base system to develop assignments that can give students hands-on experience with those theories taught in class. For example, when teaching `Set-UID` concept of `Unix`, we developed an assignment for students to play with this security mechanism, figure out why it is needed, and understand how it is implemented.

We have developed two types of assignments: small assignments and comprehensive assignments. Each small assignment focuses on one specific concept, such as `Set-UID` and access control. These assignments are usually small; they do not need much programming, and take only one week or two; therefore we can have several small projects to cover a variety of concepts in system security. However, being able to deal with each individual concept is not enough, students need to learn how to put them together. We have developed comprehensive assignments, which cover a number of concepts in one assignment. They are ideal candidates for final projects.

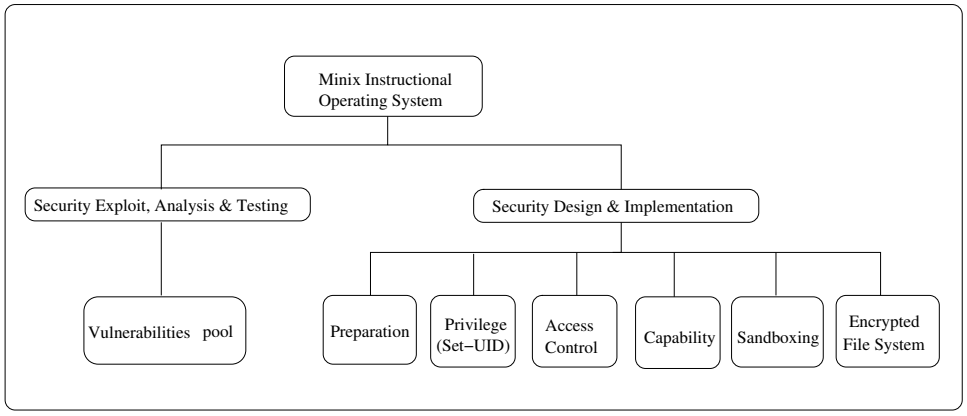

Figure 1. Overview of course projects based on `Minix`.

2.3 Course Prerequisites

Because this course focus on system security, we require students to have appropriate system background. Students taking the course are expected to have taken the graduate-level operating systems. They should be proficient in C programming.

3 Design of Course Projects

The goal of our projects is to provide a set of exercises for students to practice their *security design, implementation, analysis, testing, and operation* skills. Using the `Minix` instructional operating system, we designed two classes of projects, one focusing on design and implementation of security mechanisms, and the other focusing on security analysis and testing. The overview of our projects is depicted in Figure 1.

Design and Implementation. In our computer security class, we aim at covering a number of important security mechanisms, such as *Privilege, Authentication, Access Control, Capability, and Sandboxing*. We expect students to have first-hand experience on most of them during one semester period. However, asking students to implement a system with all of these mechanisms from scratch sounds infeasible. Using an instructional operating system, our goal becomes feasible because of the following reasons: (1) An instructional OS provides students with a structured framework upon which they can build various security mechanisms. (2) An instructional OS 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.

Some of the security mechanisms are already implemented in `Minix`, such as privilege, and access control. For some of these mechanisms, our projects are designed in a way that requires students to study and play with the existing implementation, so they can gain first-hand experience. For other existing mechanisms, we ask students to extend them and add more functionalities. For example, we ask students to extend the `Minix`'s abbreviated access control list mechanism to support full access control lists. Several security mechanisms that we cover in class do not exist in `Minix`, such as capability and encrypted file system. For them, we designed course projects that ask student to implement these mechanisms in `Minix`. To make the tasks doable with 2-3 weeks, the security mechanisms are simplified compared to those implemented in an real operating system.

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 systems. One way to do this is to give them a vulnerable system, such as older versions of Windows 2000 or Linux, and ask them to find security flaws in those systems. Although these systems contain many vulnerabilities, identifying and exploiting them is not a trivial task even for seasoned system administrators, much less students who have just learned the basic skills.

We have created a pool of vulnerable components for `Minix`, with some in the application layer and some in the kernel layer. The vulnerabilities we choose reflect vulnerabilities in the real world. They include buffer-overflow errors, race condition errors, sym-link errors, input validation errors, authentication errors, domain errors, and design errors [9].

Instructors can choose the vulnerable components they like and inject them into `Minix`. The flawed `Minix` system is then given to students, who need to find those vulnerabilities and exploit them. Before starting these exercises, 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.

3.1 Why choose `Minix`?

Before we decided to use `Minix`, we have investigated a number of alternatives. We have the following criteria in mind when choosing an operating system as the base of our courseware:

1. *Source code availability.* Because the system security course involves implementation of system security mechanisms, studying the source code is important for the learning process.

2. *Complete but not complex.* The OS should provide an sufficient infrastructure to students. Students should be able to immediately see how their implementation behaves without having to build the security-irrelevant components to make the whole system work. However, the OS should not be too complex; otherwise students need to spend much time in understanding the underlying system.

3. *Modularized.* The security modules in the system should be highly modularized, so that they can be modified or replaced independently.

4. *No need for superuser privilege.* It is preferable for students to carry out lab assignments in a general computing environment using normal user accounts, as opposed to in a dedicated computing environment using superuser privileges.

A complete featured OS like `Linux` seems a good candidate because their completeness. However, if we choose such an operating system, the students will take considerable amount of time to understand the functionality of the OS and thus lose focus on security. To overcome this drawback, many operating system courses use simplified operating systems, such as `Xinu`, `Nachos` and `Minix`, for educational purposes. We adopted a similar practice.

Most computer security course projects require the administrator/superuser privilege, which can jeopardize the security of the security experiment. With the superuser privilege, students can have complete control over the experimental domain. A malicious students might use it to gain unwanted access to other people's accounts. Even if all students are well behaved, they might accidentally introduce security holes into the system because of the lack of system administrating experience. Some universities do give students the superuser privilege for this type of projects, but the computers have to be restricted to an isolated environment. Although this approach has been widely used in

Table 1. A comparison of various operating systems.

		Source Code Availability	Complete	Complex	Superuser Privilege	Modularized
Instructional OS	Minix	Yes	Yes	No	No	Yes
	Nachos	Yes	Partial	No	No	Yes
	Xinu	Yes	Yes	No	Yes	Yes
Commercial OS	Linux	Yes	Yes	Yes	Yes	Yes
	BSD	Yes	Yes	Yes	Yes	Yes
	SunOS	No	Yes	Yes	Yes	Yes
	Windows	No	Yes	Yes	Yes	Yes

practice, it requires high cost for lab setting up and management. We choose a different approach: to enable students to build and run the operating system without giving the superuser privilege.

We chose `Minix` instructional operating system as our base system for three reasons: first, `Minix` is complete comparing to other unix-style instructional OS's; second, `Minix` can run on the `Solaris` systems as a non-privileged process; third, `Minix` is small and easy to understand. Table 1 compares the pros and cons for using different OS's as the base of our courseware.

3.2 Introduction to the Minix operating system

`Minix` is a `Unix` operating system, and its name came from "mini `Unix`". As an instructional operating system, `Minix` system is designed to be small and simple. It only has about 15,000 lines of codes, which are publicly available at [10]. A textbook was also written by Tanenbaum to explain how `Minix` works [5]. Students meeting the prerequisites can understand this operating system within a short period of time. `Minix` system has a high modular structure, which makes it not only easy to understand, but also easy for students to extend and modify.

`Minix` was originally developed as a real operating system, running directly on Intel x86 machines. Later on, Ashton ported `Minix` to run on the SUN Solaris systems as a non-privileged process [11].

4 Course Projects

4.1 Laboratories Setup

We use `Minix` on Solaris in our course. All of the laboratory 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 `Minix` system, `Minix` poses no special requirements on the general `Solaris` computing environment.

The `Minix` operating system can also be installed on simulated environments like *VMware* [12], *Bochs* [13] and so on. Installing the operating system on *VMware* is not a difficult process, and no superuser privilege is needed to run `Minix` on *VMware*. Therefore, this could be another installation option. Both approaches can be used in our laboratory designs. However, we preferred to use the Solaris approach, so students do not need to buy the *VMware* license or use free-wares that are not stabilized yet.

We have designed a variety of course projects on `Minix`. Depending on the course schedule and the students' familiarity with `Unix` and their proficiency in C programming, instructors might

want to choose a subset of the projects we designed. Currently, we are still developing more assignments, and we will also solicit contributions from other people. Our goal is to create a pool of lab assignments, such that different instructors can choose the subset to meet the requirements of their syllabi.

4.2 Preparation

In this warm-up project, students get familiar with the `Minix` operating system, such as installing and compiling the `Minix` OS, conducting simple administration tasks (e.g. adding/removing users), and learning to use/modify some common utilities. More importantly, we want students to understand the `Minix` kernel. For our system security course, students just need to understand in detail system calls, file systems, the data structure of `i-node` and `process table`. They do not need to study non-security modules such as process scheduling and memory management. Students meeting the prerequisites should be comfortable with the `Minix` environment in two to three weeks.

The following is a list of sample tasks we used. In reality, instructor can choose different tasks to achieve the same goals:

- Compile and install `Minix`, then add three user accounts to the system.
- Change the password verification procedure, such that a user is blocked for 15 minutes after three failed trials.
- Implement system calls to enable users to print out attributes in `i-node` and `process table`. Appropriate security checking should be implemented to ensure that a user cannot steal information from other accounts.

Our experiments show that it is better to guide students to conduct the above tasks in one or two lab sessions, in which a teaching assistant can provide immediate helps. These lab sessions are extremely necessary when students have significantly different backgrounds.

4.3 `Set-UID` Programs

`Set-UID` is an important security concept in `Unix` operating systems. It is a good example to show students how privileges are escalated in a system. In this project, students learn the `Set-UID` concept and its implementation. Students also learn how an attacker can escalate its privileges via exploiting a vulnerable `Set-UID` program.

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

This project is quite straightforward. On average it takes students one week to finish.

4.4 Access Control List

Access control is an important security mechanism implemented in many systems. It can be classified as Discretionary Access Control and Mandatory Access Control (MAC). In DAC systems,

the owner of an object can decide its security properties (e.g., who can read this file?); while in MAC systems, the security properties are determined and controlled by only a security manager. Access permissions can be represented on a per object basis (i.e. who can do what operations on an object); this is called *Access Control Lists*. Permissions can also be represented on a per subject (principal) basis (i.e. what operations on what objects the subject can do); this is called *Capabilities*. This project focuses on access control lists.

The goal of this project is two-fold: (1) to get first-hand experience with DAC and (2) to be able to implement DAC. `Minix` already has an implementation of abbreviated ACL; namely the access control is based on three classes: owner, group, and others. Students need to extend this abbreviated ACL to a full ACL, i.e., a user can assign a specific access right to any single user. On average students need about 2-3 weeks to finish this project. Students need to deal with the following issues:

- *How access control works:* Before working on their implementations, students need to understand the entire process of access control, and they need to trace the program execution to find out how the access control is conducted in `Minix`. This enhances their understanding of access control.

- *ACL representation:* Students need to think about how to represent the full ACL, how to allow ACL's to specify access permissions on a per user (principal) basis, rather than the current owner-group-other protection method. Students also need to make their representation flexible for adding and removing purposes.

- *Storing the ACLs:* This is another challenging part of the project. Students need to think where exactly they should store the access control list. The current `Minix` implementation does not seem to have a place to store the full access control list. Students need to solve this issue. A hint we give them is to use some unused entries in i-nodes or store the access control lists in separate files.

- *ACL management:* In addition to implementing the full ACL in the kernel, students also need to implement the corresponding utilities, such that users can manage the access control list of their own files.

4.5 Capability

Capability is another important concept in computer security. The goal of this project is to help students understand the concept of capability. We defined a set of capabilities in this project, with each capability representing whether a process can invoke a specific system call. Students need to implement these capabilities in `Minix`. Specifically, their capability mechanism should be able to achieve the following functionalities: (1) Permission granting based on capability. (2) Capability copying: A process should be able to copy its capabilities to another process. (3) Capability reduction/restoration: A process should be able to amplify or reduce its current capabilities. For example, a process can temporarily remove its own `Set-UID` capability, but later can add it back. Of course, a process cannot assign a new capability to itself. (4) Capability revocation: Root should be able to revoke capabilities from processes.

In this project, students need to take care of the following issues:

- *Capability List Representation:* Students need to think about how to represent the set of defined capabilities. They also need to think how they can associate capabilities with each

process. The final representation should conveniently support the required functionalities (e.g. copying, removing etc).

- *Storing the Capabilities:* This is another challenging part of the project where students need to think where capabilities should be stored. One option is to add an entry to the process table to store the capabilities. A potential issue is how feasible it is to extend the process table (note that the process table is a kernel data structure used by many other components).

- *Capability Revocation:* Students need to think about how to revoke an object's capability. They must be careful not to introduce vulnerabilities in this part.

- *Capability Management:* Students need to take care of two types of users, normal and super users. They need to consider the following issues: how they manage these two types of users, and what functionalities are associated with each of them.

This project enhanced the students' understanding of the capability concept. At the beginning, most students had trouble mapping the capability concept to the real world. We did not tell the students how the capability should be implemented, but to ask them to design their own capability mechanisms. This requires them to figure out how the capabilities should be represented in the system, where to store the capabilities, how the system can use the capability to conduct access control, etc. Once students have figured out all of these issues, the implementation becomes relatively easy; therefore the amount of coding for this project is not significant, and students are able to accomplish the task within two weeks. Had it not been for `Minix`, students would need to spend a lot of time implementing a meaningful system where the effect of the capability can be demonstrated.

We encouraged students to design some other features beyond the basic requirements. Students were highly motivated, some implemented a more generic capability-based access control mechanism than the required one, and some allow new capabilities to be defined by the superuser.

4.6 Sandbox

A sandbox is an environment in which the actions of an untrusted process are restricted according to a security policy [14]. Such restriction protects the system from untrusted applications. In `Unix`, `chroot` can be used to achieve a simple sandbox.

The instruction "`chroot` *newroot cmd*" causes *cmd* to be executed relative to *newroot*, i.e., the root directory is changed to *newroot* for *cmd* and any of its child processes. Any program running within this sandbox can only access files within the subdirectory of *newroot*.

Some `Unix` systems allow normal user to run `chroot` sandbox (just make `chroot` a `Set-UID` program). However, this can introduce a serious problem: malicious users may create a login environment with their own shadow file and passwd file under *newroot*, which will help them gain a root shell. Once getting that privilege, they can create a `Set-UID` shell program which allows them to use after exiting the sandbox. The attack is described in the following:

```
test $ mkdir /tmp/etc
test $ echo root::0:0::/:/bin/sh > /tmp/etc/passwd
test $ mkdir /tmp/bin
test $ cp /bin/sh /tmp/bin/sh
test $ cp /bin/chmod /tmp/bin/chmod
test $ chroot /tmp /bin/login  (login as root with no password)
root # chmod 4755 /bin/sh (change shell to Set-UID)
```

```
root # exit
test $ cd /tmp/bin
test $ ./sh
root # (get root shell in real system)
```

One of the goals of this project is to let students find out this vulnerability with some provided clues. Students need to implement attack procedures and demonstrate how to take advantage of the vulnerability to gain root privileges. This is an efficient way for students to enhance their understanding on security hole in kernel level.

To fix the above vulnerability, the best way is to disallow normal user from using `chroot`. However, normal users will not be able to take advantage of the sandbox. We ask students to extend the current `chroot` such that the program is safe to be used by normal users.

We suggest students to design a security policy for this sandbox. Sandbox security policy defines a set of permissions and restrictions that a program must obey while running. For example, the policy can define whether a program is permitted to read files or connect to the Internet. Any program attempting to violate the security policy will be blocked. Students need to consider a number of issues, including how to define policy, where to save policy, when it should be read in, and how to secure the policy file. Students should be able to finish this project within 2-3 weeks.

4.7 Encrypted File System

Non-encrypted file system stores plain text on disks, so if the disk is stolen, information on it can be disclosed. An Encrypted File System (EFS) solves this problem by encrypting the file system, such that only users who knows the encryption keys can access the files. The primary benefit of EFS is to defend against unauthorized access. The encryption/decryption operations should be transparent to users. Implementing EFS requires students to combine techniques such as encryption, key management, authentication, access control, and security in OS kernels and file systems; therefore this project is a comprehensive project. We give this project as a final project.

`Minix` system has a complete file system, so students can build the EFS on top of it. As we mentioned before, `Minix` file system is reasonably easy to understand; students can start building their own EFS after they understand how the file system works.

This project is a good candidate for the final comprehensive project because it covers a variety of security-related concepts and properties:

- *User Transparency:* The main challenge of this project is how to make EFS transparent. If the transparency is not a issue, then students can easily implement a set of encryption/decryption utilities, and users need to use those utilities to encrypt/decrypt their files *manually*. The transparency means that the encryption/decryption should be performed on the fly, while users are reading/writing their files. To achieve the transparency, students need to modify the system calls related to the reading and writing. They need to insert the encryption algorithms into the proper positions in those system calls.

- *Key management:* Another challenge of this project is the key management, namely how and where the encryption keys should be stored, how the keys should be protected, changed, and revoked. We have seen different designs from students. For example, regarding the key storage problem, some students store the key (encrypted) in a file, and some store it in the i-node of the encrypted file. We also found out that some students mistakenly save the plain-text key on the disk, which defeats the whole purpose of the EFS.

- *Authentication:* How to decide whether a user can access the encrypted file system or not? This part of the project not only teach students the authentication purpose, more importantly, it teaches students an important lesson about the tradeoff between the usability and the security. Some students' projects require users to authenticate themselves each time when they access a file in EFS; some conduct just one authentication when the users mount the EFS (a good implementation in our opinion); some conduct the authentication during the login. During their demos, we point out the advantages and disadvantages of their designs, so they can evaluate their own designs.

- *Using encryption and hashing algorithms:* Although students are provided with codes for encryption and hashing algorithms, they still need to learn how to use it correctly. Because AES is a block cipher, students need to deal with the issues related the block and padding; otherwise, their reading/writing system calls might not function correctly.

- *Security analysis:* After most of the students have finished their designs, we give them several incorrect designs that we have encountered in the past, and we asked them to find out whether those designs are secure or not; if not, how to break those EFS.

Project Simplification

For students who do not have sufficient background in operating system kernel programming, we need to customize our projects for them. We divide the EFS project into three projects:

1. Project 1: Encryption Algorithms. This project gets students familiar with the AES algorithm. Students need to implement a user-level program to encrypt and decrypt files.

2. Project 2: Kernel modification. The second project asks students to modify the corresponding system calls, such that some special files are always read/write using encryption. However, to simply this project, we ask them to always use a fixed key for the encryption. The key can be hard-coded in their programs.

3. Project 3: Key Management. This project deals with the key management issue that is intentionally left off in the previous project. Students now need to find a place to store the key; they need to make decision on whether to use the same key for all the files or one key for each file; they also need to deal with the authentication issues, etc.

4.8 Vulnerability Analysis

Vulnerability analysis strengthens the system security by identifying and analyzing security flaws in computer systems. This project intends to expose students to such an critical approach. We have two goals in this project: The first goal is to let students gain first-hand experience on software vulnerabilities, be familiar with a list of common security flaws, and understand how a seemly-not-so-harmful flaw in a program can become a risk to a system. The second goal is to give students opportunities to practice their vulnerability analysis and testing skills. Students can learn a number of methodologies from the class, such as vulnerability hypothesis, penetration testing methodology, code inspection techniques, and blackbox and whitebox testing [15]. They need to practice these methodologies in this project.

To achieve our goals, we modify the `Minix` source codes and intentionally introduce a set of vulnerabilities. We call these vulnerabilities the *injected vulnerabilities*. The revised `Minix` system

is then given to students. The students are given some hints, such as a list of possible vulnerabilities, the possible locations of the vulnerable programs, etc. Their task is to find out and verify these vulnerabilities.

The injected vulnerabilities cover a wide spectrum of vulnerabilities, such as buffer overflow, race condition, security holes in the access control mechanisms, security holes in `Set-UID` programs, information leakage, and denial of service. These vulnerabilities reflect system flaws caused by incorrect design, implementation, and configuration. All these vulnerabilities are collected from real commercial Unix operating systems, such as SUNOS, HP-UNIX and Linux, and are then ported to `Minix`. We have ported nine vulnerabilities so far, with six in the user level and three in the kernel level. We will port other typical vulnerabilities to `Minix` in the future.

Students in this project need to accomplish the following tasks:

- *Identify vulnerabilities*. This is a warm-up practice to help students get familiar with vulnerability living environment.
- *Exploit vulnerabilities*. This is a challenging and interesting part of the project in which students write attack programs aiming at these vulnerabilities. Demonstration is needed to show what unauthorized privilege can be obtained.
- *Fix vulnerabilities*. Students need to design solutions to eliminate or remedy the identified vulnerabilities.

5 Experiences and Lessons

We did a teaching experiment in the 2002 spring semester when we taught the graduate-level computer security course at Syracuse University. At that time, we asked students to add certain specific security mechanisms to `Minix`. We only give students 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 project 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 system 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.

When we taught the course again in Spring 2003, we provided students with sufficient information on how `Minix` works, and we added a lecture to introduce `Minix`. As a result, students had gotten familiar with `Minix` within the first three weeks, and were ready for the projects we had designed for them. The same degree of familiarity took students half of a semester previously due to the lack of information.

In our first experiment in 2002, the requirements of each project were not tailored to a scope appropriate for 2-3 weeks. During the last three years' experiments, we simplified those requirements. In 2004 semester, we successfully assigned four projects in one semester, including the `Set-UID` project, capability project, access control project, and the comprehensive encrypted file system project. However, we are still unable to assign the vulnerability project due to the lack of time. We will further improve our strategy in the coming 2005 Spring semester.

During the last three years, we have also learned the following lessons:

- *Preparation:* From our experience, the preparation project is crucial to the success of the subsequent assignments. Some students who overlooked this assignment find themselves in

trouble later. In fact, when we used the proposed approach at the first time, we did not give students this assignment because we thought it was not necessary. As a result, students later spent a great deal of time in figuring out how to achieve the tasks in this assignment. Most of the students told us that they spent 80% of their time to get familiar with the system. Once they knew how `Minix` works, they can spend short time to finish the required task. Therefore, when we use the approach again, we used several lectures to inform students the necessary materials, and ask the TA to devote significant amount of time to help the students finish this assignment. The preparation part is extremely important. If students fail 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 our 2003 course with that of the students in 2002. We plan to integrate the materials related to `Minix` into the lecture, so students can be prepared better.

- *Background knowledge:* We also realized that some students in the class are not familiar with the `Unix` environment because they have been using Windows most of the time. This brings some challenges 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 obstacle.

- *Cheating:* Cheating did occur, especially on the final encrypted file system project. We now have a list of questions that we will ask during student's demonstrations. They not only help us evaluate students' projects, but also are quite effective so far in identifying cheatings. Example of questions include "where do you save keys and why?", "can your implementation work on large files? and how did you handle that?", etc. Students who simply copy others' implementation will be most likely unable to answer these questions.

6 Conclusion and Future Work

We have described a laboratory design for our graduate-level computer security course. Our approach is intrigued by the successful practice in operating system and network courses education. In our approach, we use `Minix` instructional operating system as the basis of our laboratory; in design-oriented laboratory projects, students add a specific security mechanism to the system; in analysis-oriented laboratory projects, students identify, exploit, and fix vulnerabilities in `Minix`. Because of the desirable properties of `Minix`, our laboratory projects can be finished within a reasonable amount of time and in a general computing environment without using superuser privileges. We have designed a series of laboratory projects based on `Minix`, and have experimented with our approach for the last three years. The experience obtained is encouraging, and students in our class have shown great interest in the course and the projects.

We will continue experimenting and perfecting our approach. More importantly, we will work on making this laboratory approach easy to be adopted by other people. This requires us to provide detailed documentations, instructions, and a pool of different projects covering a wide range of security concepts.

References

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

[2] J. M. D. Hill, C. A. C. 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*, Charlotte, NC, USA, February 2001, pp. 36–40.

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

[4] 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*, Charlotte, NC, USA, February 2001, pp. 31–35.

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

[6] W. A. Christopher, S. J. Procter, and T. E. Anderson, "The nachos instructional operating system," in *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, USA, January, 25-29 1993, pp. 481–489, available at `http://http.cs.berkeley.edu/~tea/nachos`.

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

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

[9] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Computing Surveys*, vol. 26, no. 3, pp. 211–254, September 1994.

[10] A. Tanenbaum, "http://www.cs.vu.nl/~ast/minix.html."

[11] P. Ashton, "Smx–the solaris port of minix," 1996.

[12] VMWare, "http://www.vmware.com."

[13] Bochs, "http://bochs.sourceforge.net."

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

[15] C. Pfleeger, S. Pfleeger, and M. Theofanos, "A methodology for penetration testing," *Computers and Security*, vol. 8, no. 7, pp. 613–620, 1989.