ASEE 2022 ANNUAL CONFERENCE
Excellence Through Diversity
MINNEAPOLIS, MINNESOTA, JUNE 26TH-29TH, 2022 ASEE

Paper ID #37982

Mini-projects based Cybersecurity Modules for an Operating System Course using xv6

Jansen Tan (Purdue University Northwest)

Divya Ravindra (Purdue University Northwest)

Quamar Niyaz

Quamar Niyaz received the B.S. and M.S. degrees in computer science and engineering from Aligarh Muslim University, in 2009 and 2013, respectively, and the Ph.D. degree from The University of Toledo, in 2017. He has been an Assistant Professor in computer engineering with the ECE Department, Purdue University Northwest, since 2017. He has published papers in the areas of computer and networks security, applied machine learning, and cybersecurity education. His research has been sponsored by the National Science Foundation.

Xiaoli Yang

Dr. Xiaoli (Lucy) Yang is currently the chair and professor of the Department of Computer Science and Engineering at Fairfield University. Dr. Yang's main research interests include virtual/augmented reality, , cybersecurity education, machine learning applications, and software engineering. She has published more than 80 papers in journals and refereed international conference proceedings, and one book by Springer. Dr. Yang has received grants from NSF-National Science Foundation, Indiana Commission of Higher Education, Northwest Indiana Computational Grid Grant, and NSERC-Natural Sciences and Engineering Research Council of Canada.

Sidike Paheding

Ahmad Y Javaid (Dr.)

Ahmad Y. Javaid received his B.Tech. (Hons.) Degree in Computer Engineering from Aligarh Muslim University, India in 2008. He received his Ph.D. degree from The University of Toledo in 2015 along with the prestigious University Fellowship Award. Previously, he worked for two years as a Scientist Fellow in the Ministry of Science & Technology, Government of India. He joined the EECS Department as an Assistant Professor in Fall 2015 and is the founding director of the Paul A. Hotmer Cybersecurity and Teaming Research (CSTAR) lab. Currently, he is an Associate Professor in the same department. His research expertise focuses on application of computational intelligence to various computing domains including but not limited to education, cybersecurity, healthcare, human-machine teaming, and digital forensics. His projects have been funded by various agencies including the NSF (National Science Foundation), AFRL (Air Force Research Lab), NASA-JPL, Department of Energy, and the State of Ohio.

Mini-projects based Cybersecurity Modules for an Operating System Course using xv6

Jansen Tan[1], Divya Ravindra[1], Quamar Niyaz[1], Xiaoli Yang[2], Ahmad Y Javaid[3], Sidike Paheding[4]
[1]ECE Department, Purdue University Northwest, Hammond, IN 46323
[2]CSE Department, Fairfield University, Fairfield, 06824, CT, USA
[3]EECS Department, The University of Toledo, Toledo, OH 43606
[4]Applied Computing, Michigan Technological University, Houghton, MI 49931

1. Introduction

Cybersecurity is critical nowadays with the increased reliance on computing systems and technology. The cyberattacks in the past mostly damaged digital information leading to financial or reputational loss, but now they are targeting physical infrastructure as well [1, 2, 3]. The attackers attempt to exploit vulnerabilities at every level in the targeted computing systems, i.e. hardware, software, and the system environment to compromise their security. An operating system (OS) is an essential system software in multitasking computing systems that resides at the low-level after hardware and manages system resources for applications running simultaneously on top of it. The security of an OS is critical due to its installation on every computer unlike other software, which may or may not be installed in a particular system. If any security flaw exists in the OS, it will affect all the applications running on top of it and will make them vulnerable [4]. Although addressing security issues in OS development has been a key requirement for a long time, still many vulnerabilities, such as memory exploits and privilege escalation, are discovered over time. The reasons for the occurrence of these vulnerabilities are the complex OS code and its support for concurrent trusted/non-trusted processes. Another significant issue is the lack of discussion on security aspects when OS courses are taught in computer science (CS) and computer engineering (CE) curriculum. The emphasis is given to process scheduling, memory management, concurrency control, and I/O handling. The discussion on security is deferred for security courses, which are offered at senior undergraduate or graduate level. This approach limits the practices of secure system development that encourages inclusion of security measures at the inception stage of system development. Therefore, it is important for CS/CE students that they should have an exposure to OS-related security concepts while they are taking a course on it. Later on, they can sharpen this knowledge in other security courses or in their professional work environment.

Many pedagogical OSs have been developed to teach OS courses through hands-on lab exercises. Several universities have developed lab assignments using them for OS design concepts. However, little efforts have been made towards the development of security-related labs in the OS courses. Although few independent projects have been found that will be discussed in the related works section, they do not offer proper documentation or their implementation is quite naive compared to production-level OS. To bridge this gap, we develop mini-project based modules for security concepts that instructors can adopt in their OS courses. These modules are built for xv6 – a modern pedagogical OS [5] and they focus on security along with the OS development. The concepts covered in these modules include authorization, access control, and address space layout randomization.

Towards this, the outline of the rest of the paper is as follows. The methodology of the project is discussed in Section 2. Section 3 provides an overview of each module. Section 4 compares the implementation of our developed modules to similar functions in other OSs. Related work is discussed in Section 5. Finally, the paper is concluded with an insight for future work in Section 6.

2. Methodology

The project modules have been designed for undergraduate CS/CE students who are enrolled in an OS course or have taken it. The students must have a background in C programming and familiarity with the commands in Unix-like systems. These modules are designed in a way that they could be completed within the course of a semester following the documentation created for each module. We chose xv6 as the platform on which these project modules are based for a number of reasons. Xv6 is publicly available open-source instructional OS to build and modify, which avoids issues with cost and licensing. It is lightweight and the source code only occupies half a megabyte of storage. In addition, xv6 compilation takes at most a few seconds on modern machines, and running it on an emulator (e.g. QEMU) is smooth as well. It enjoys a long history of refinement since its inception in 2006. For example, the source code in the current version (i.e. xv6-riscv) is organized more cleanly than the previous version [6]. In the older version, all the source files are located in the project's root directory. In the latest version, files are organized into three sub-directories: `kernel`, `user`, and `mkfs`. The documentation for xv6 is also publicly available in the form an accompanying book.

Compared to other pedagogical candidate OSs such as Minix [7], Xinu [8], and Unix V6 [9], xv6 is the preferred choice for developing these project modules. The other platforms may have advantages similar to that of xv6, but do not share all of the advantages of xv6. For example, Minix, today is developed to be a production-quality OS used in real systems. The modern version of Minix, called Minix 3, has become much larger in storage and grown in complexity compared to earlier versions, thus making it tedious for pedagogical purposes. Earlier versions of Minix, while retaining their pedagogical intent, are suffering from old age that make building and booting earlier Minix a confusing and unreliable process compared to xv6 [10, 11]. Although there is a recent project that simplifies the Minix boot procedure, it does so in a "quick-and-dirty" way [12]. Xinu does not have a freely available accompanying book, which makes it less attractive to use. Early versions of Unix, such as V6, are very old and hard to build with recent build tools. They are also not properly organized, somewhat complex, and the accompanying documentation in the educational context is less user-friendly than that of xv6. Table 1 summarizes the comparison of these candidate platforms.

Table 1: Comparison of candidate pedagogical OSs

Candidate OS	**Feature complexity**	**Build process complexity**	**Documentation**
xv6	Low	Low	Good
Minix 1, 2	Low	High	Good
Minix 3	High	High	Good
Early Unix	Low	High	Poor
Xinu	Low	Low/High	Not publicly accessible

The modules that we selected for the development have been adopted from the security topics discussed in a popular OS textbook – Operating Systems: Three Easy Pieces [4]. For implementation, attempts have been made to follow modern operating systems standards as best as possible while keeping pedagogical appropriateness. To do so, various resources were referenced including Unix and Linux manual **(man)** pages to get an idea of current implementations of a few systems such as the header file `pwd.h`.

3. Overview of Modules

We developed three modules as mini-projects for adoption in an OS course. These modules include the implementation of authentication, file access control, and address space randomization. Each module is accompanied by proper documentation that details how it can be implemented. The documentation provides background on the covered topic to create a sense of direction and then a goal to provide an expectation on the output. The module documentation does not attempt to detail how to manage other aspects of the project, such as installation, debugging, and low-level OS mechanics. However, these are certainly critical aspects of the modules. Therefore, these details are placed in an appendix aside from the module documentation. Each module documentation references relevant sections of the appendix whenever appropriate. A Github repository[1] containing branches of riscv-branch of xv6-riscv repository has been created. These branches named as `auth-student`, `access-control-student`, and `aslr-student` contain skeleton code for students' implementation of these modules. Implementation of each module is discussed as follows:

3.1 Authentication module

This module allows students to implement a password-based user authentication system into xv6. The module guides the implementation such that students will be aware of current OS standards regarding password-based user authentication, and follow the standards throughout the implementation. In this module, a student will implement the user account abstraction into xv6. The implementation conforms to POSIX specification for the `pwd.h` header file. The student will create the `struct passwd` structure and some of the `pwd.h` functions specified by POSIX, such as `getpwent()`. Then, a `useradd` program will be created. For simplicity, this program will combine functionality of `useradd` and `passwd` programs in the Linux system. The student will then implement process ownership into xv6. The kernel's process control structure `struct proc` is modified to include `uid` and `gid` fields and various kernel functions and system calls are implemented to interact with the new fields. The student will then create a simple `whoami` program that can be used to test the overall authentication implementation. The student will then create the login program. The login program synthesizes the functions created in the previous steps. The user account abstraction is used to authenticate against given credentials. After successful authentication, the current process is set to be owned by the user, and then the process executes the shell. The student will then modify the `init` process to start the login program on boot instead of the shell program.

The `auth-student` branch contains skeleton code to facilitate the writing process. A header file for user account abstraction is provided at `user/pwd.h`. The header file contains all needed function prototypes and an empty declaration of the `struct passwd` structure. The corresponding file `user/pwd.c` is also provided containing all needed function declarations and comments guiding the writing process. The programs `user/useradd.c`, `user/whoami.c`, and

[1] https://github.com/jansenmtan/xv6-riscv

user/login.c are provided containing all necessary directives, function declarations, and comments. A test program is provided at user/pwdtests.c to test the functionality of the module.

3.2 Access Control Module

This module allows students to implement Unix-style access control into xv6. This section is identical to the corresponding section in the authentication module. The kernel's process control structure struct proc is modified to include uid and gid fields; and various kernel functions and system calls are created to interact with the fields. The student will then implement file ownership. Inode structures are modified to include the uid and gid ownership bits. To allow the ownership bits to be changed, the chown kernel function and chown program are created. The student will then implement access control lists. Inode structures are modified to include Unix-like permission bits. The student will then implement access authorization. An access kernel function is created to verify that a given file is either readable, writable, or executable. Various kernel functions that interact with files are modified to enforce access authorization. The student will implement default ownership and mode for newly created files.

The access-control-student branch contains skeleton code to facilitate the writing process. Comments are placed in various kernel programs, such as at kernel/file.c, kernel/proc.c, and kernel/sysfile.c. Permission bit masks are added into the header file at kernel/fcntl.h. An octal conversion specifier is also written into the user/printf.c program to help with printing permission bits.

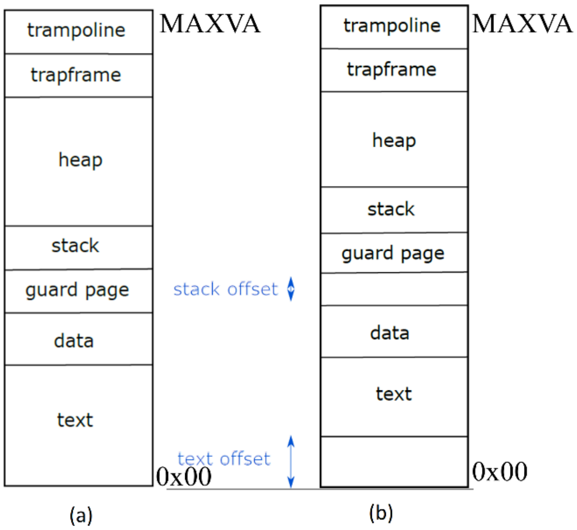


Figure 1: Memory address layout with disabled ASLR (a) and enabled ASLR (b).

3.3 Memory randomization module

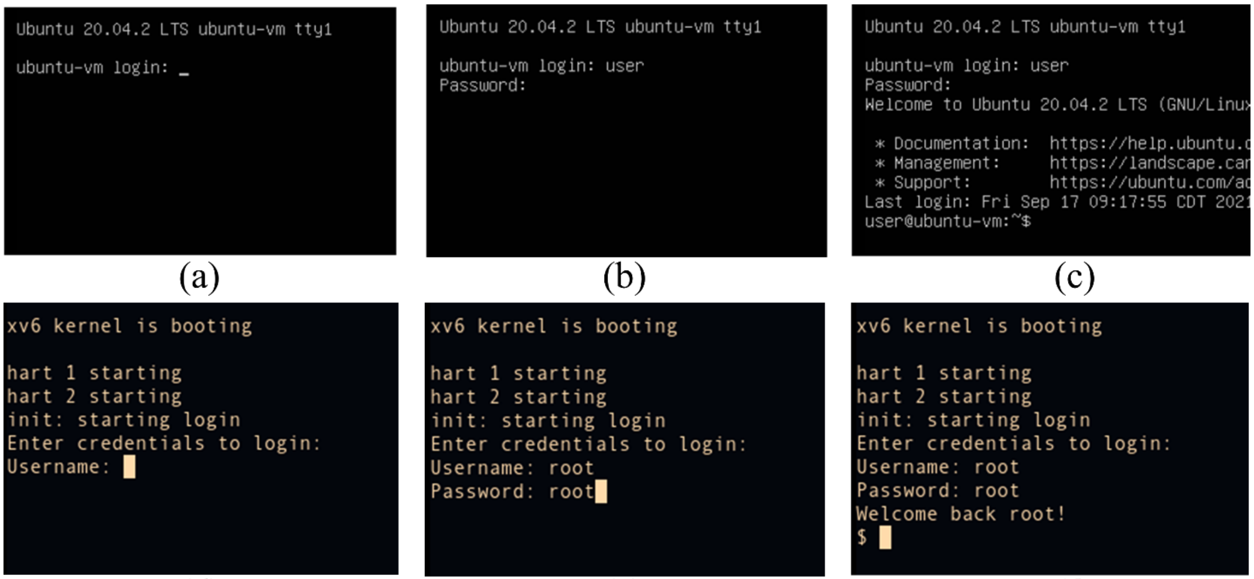
This module allows students to implement a basic version of Address Space Layout Randomization (ASLR) technique into xv6. In this module, a student will create a random number generator. The student will then modify the executable file loader. Two memory segments within a process will be modified. The program text segment will be modified to have a random offset from the beginning of the virtual address space. The stack segment will be modified to have a random offset from the end of the program text and data segments. The changes made to the user address layout can be seen in Figure 1 (b) and compared to the unchanged layout in Figure 1 (a). The student will then implement configuration that allows ASLR to be configured on or off.

The aslr-student branch contains skeleton code to facilitate the writing process. Comments are placed in a few kernel programs and follow the process written in the ASLR module documentation. A program aslrtest.c is provided at user directory, which allows checking for layout randomization at runtime.

4. Validation of Developed Modules

Our implementation of these modules has been tested to produce behavior similar to that on corresponding production grade systems, such as Linux.

4.1 Authentication module



```
Ubuntu 20.04.2 LTS ubuntu-vm tty1

ubuntu-vm login: _
```

(a)

```
Ubuntu 20.04.2 LTS ubuntu-vm tty1

ubuntu-vm login: user
Password:
```

(b)

```
Ubuntu 20.04.2 LTS ubuntu-vm tty1

ubuntu-vm login: user
Password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linu

 * Documentation:  https://help.ubuntu.c
 * Management:     https://landscape.car
 * Support:        https://ubuntu.com/ac
Last login: Fri Sep 17 09:17:55 CDT 2021
user@ubuntu-vm:~$
```

(c)

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting login
Enter credentials to login:
Username:
```

(d)

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting login
Enter credentials to login:
Username: root
Password: root
```

(e)

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting login
Enter credentials to login:
Username: root
Password: root
Welcome back root!
$
```

(f)

Figure 2: Comparison of authentication process between Linux and our implementation in xv6.

In various Linux distributions, the authentication process usually happens as demonstrated in Figure 2. The `init` system starts the login interface, which is the first interactive interface the user encounters after booting the system. The login interface prompts the user to enter their login credentials, as shown in Figure 2 (a). After the user inputs their credentials as shown in Figure 2 (b), the authentication process starts. Upon successful authentication, the user is able to interact with the machine in a meaningful way. The form of interaction varies depending on the computing environment. For example, on a machine with a desktop environment, the user is able to interact with the desktop. On a machine with no desktop environment, typically, the user is able to input commands into a shell, as depicted in Figure 2 (c). In our implementation, the authentication process is as follows. After booting xv6, the user first encounters the login prompt shown in Figure 2 (d). If there are no users present in the user account database, then login prompts the user to create the first user (i.e., the root user) and calls `useradd` to do so. Otherwise, login prompts the user to enter their login credentials, as shown in Figure 2 (e). Upon successful authentication, the user is able to input commands into the shell as depicted in Figure 2 (f).

4.2 Access control module

In most Unix-like operating systems, access control is managed with a number of file permission bits and file ownership [13]. Access control in Unix-like operating systems has a precedence feature. This feature gives precedence to the more relevant group of bits in the file permission bits. For example, a file marked as 177 would not be writable by the owner, although every other user in the system would be able to write to the file, precedence checks start with the owner. In Unix-like operating systems, there are two types for access control validation, one for regular files and other for directories. Regular files must have access control enforced as such:

- Only files marked as readable should be able to have their contents read or copied.
- Only files marked as writable should be able to have their contents be modified.
- Only files marked as executable should be able to be executed.

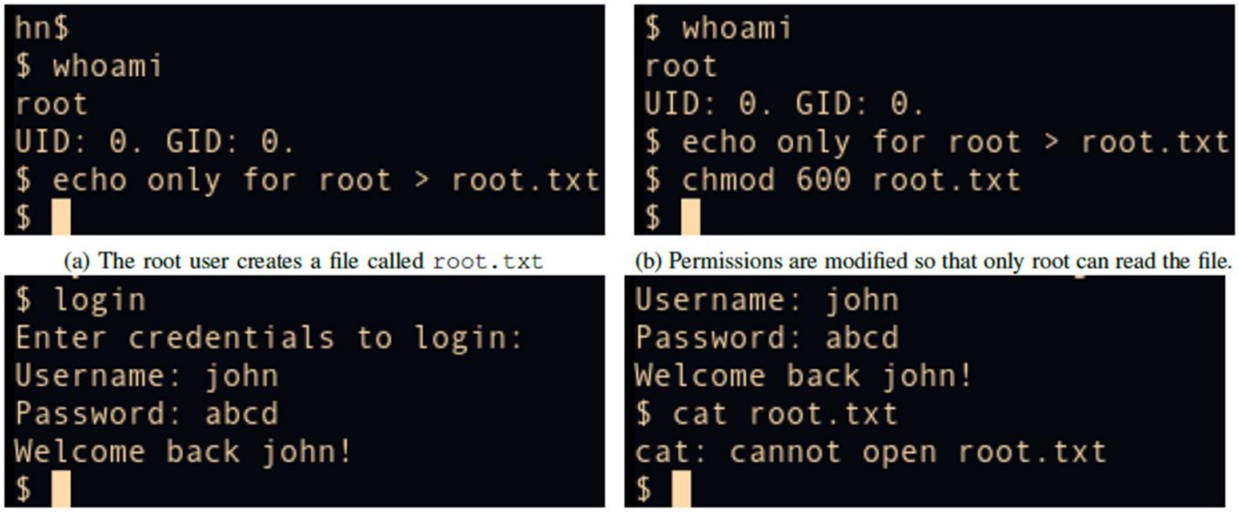
Directories must have access control enforced as such:

- Only directories marked as readable should be able to have files within it listed and copied to other directories.
- Only directories marked as writable and executable should be able to have files added or removed from within it.
- Only directories marked as executable should be able to be traversed.

In our implementation of the access control module, files may read, write, or execute by the users if one of the following conditions are met:

- The root user always has access
- The read bit for either owner, group, or others is set then the file can be read.
- The write bit for either owner, group or others is set then the file can be modified.
- The execute bit for either owner, group, or others is set then the file can be executed.

The permissions for the files can be modified either by the owner of the file or root through `chmod` command. Set-user-ID capabilities have been partially implemented in our access control module. Figure 3 shows our implementation of access control module.



```
hn$
$ whoami
root
UID: 0. GID: 0.
$ echo only for root > root.txt
$ 
```

(a) The root user creates a file called `root.txt`

```
$ whoami
root
UID: 0. GID: 0.
$ echo only for root > root.txt
$ chmod 600 root.txt
$ 
```

(b) Permissions are modified so that only root can read the file.

```
$ login
Enter credentials to login:
Username: john
Password: abcd
Welcome back john!
$ 
```

(c) Another user logs in to the system.

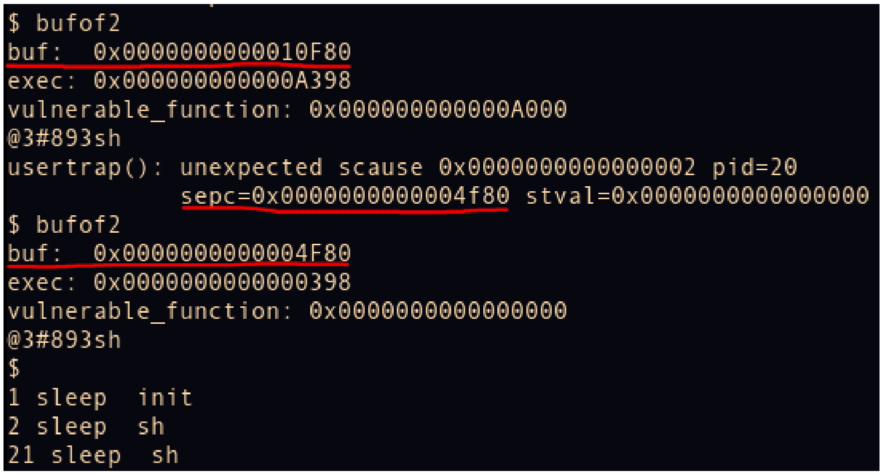
```
Username: john
Password: abcd
Welcome back john!
$ cat root.txt
cat: cannot open root.txt
$ 
```

(d) The other user cannot read the file.

Figure 3: Implementation of access control module in xv6

4.3 ASLR Module

Most modern operating systems implement some sort of address space randomization to defend against memory exploits. In Linux, ALSR currently encompasses the following features: randomization of the executable memory segments, `brk()` managed heap, `mmap()` managed memory, the user stack, and more [14, 15]. In addition, Linux provides support for global ASLR configuration and per-process ASLR configuration [16]. In our xv6 implementation, the offsets of the program text segment and user stack are randomized, as illustrated in Figure 1(a). ASLR configuration is supported globally through modifying a variable in source code. In a 64-bit Linux system, the effective entropy of memory objects ranges from 19.5 to 39 bits [16]. In particular, the executable and main stack have 27 and 35 bits of entropy, respectively [17]. Our implementation of the ASLR module only contains 4 bits of entropy for both the executable and main stack objects. In Figure 4, our ASLR implementation is shown to mitigate a buffer overflow attack. In one of the trials, ASLR mitigates an attack for 19 attempts before failing.



```
$ bufof2
buf:  0x0000000000010F80
exec: 0x000000000000A398
vulnerable_function: 0x000000000000A000
@3#893sh
usertrap(): unexpected scause 0x0000000000000002 pid=20
            sepc=0x0000000000004f80 stval=0x0000000000000000
$ bufof2
buf:  0x0000000000004F80
exec: 0x0000000000000398
vulnerable_function: 0x0000000000000000
@3#893sh
$
1 sleep  init
2 sleep  sh
21 sleep  sh
```

Figure 4: The bufof2 program reads shellcode into a vulnerable buffer to spawn a new shell. To succeed, the location of the buffer in memory must be at 0x4F80. In this demonstration, with our ASLR enabled, the attack takes 19 attempts to succeed.

5. Related Works

There are various projects, labs, and assignments created with the intention of teaching operating systems and security concepts through xv6. Many of these assignments are given in the course work at various universities and focus on topics related to OS design such as system calls, process scheduling, memory mapping, semaphore, network stacks, and interaction between computer architecture and OS [18, 19, 20, 21]. A few independent projects implement security modules similar to our modules. These projects focus mostly on the implementation and do not have robust documentation that details how to implement the features. In [22], a project is implemented for authentication and access control into the x86 version of xv6. The authentication work includes being able to log in to accounts stored in the user account database. However, there is no ability to add new users to the user account database through a user program. Modifying the user account database must be done by editing the `etc-passwd` file. The access control work includes the implementation of the `chmod` and `chown` system calls. However, access control is enforced only through one userland program: `access`. This access control implementation is appropriate for demonstration, but may not be convincing to undergraduate students; any user can run cat to print out the contents of any file containing sensitive information, such as the user account database, onto the terminal. In [23], ASLR is implemented into xv6-riscv that uses a Park-Miller random number generator to provide offsets for the program text memory segment and the stack memory segment. The heap is positioned after the stack with no random offset. ASLR is configured globally through the file `/randomize_va_space`. This project also implements support for the dynamic symbol table and relocations in an ELF. A similar independent project implements ASLR into xv6-riscv, among other features [24]. It implements virtual memory areas (VMAs) into xv6 and uses them to facilitate ASLR. A combination of kernel ticks and various mathematical operations such as exclusive-or-ing, bit shifting, and squaring are used to provide offsets for each VMA. The creation of VMAs and a heap VMA allow the kernel functions that modify the heap, `growproc()` and `sys_sbrk()` to support ASLR through simple changes. However, the introduction of VMAs into the kernel does necessitate changes to other functions in the kernel which assume a particular address layout, such as `copyin()` and `copyout()`. ASLR was intended to be an easy-to-implement fix that provided a larger benefit, so avoiding the implementation of VMAs in our work may be justified.

6. Conclusion and Future Works

We discussed the implementation of mini-project based OS modules that focus on security, and attempt to take advantage of the speed and simplicity offered by modern tools. These modules are designed to be completed within a few weeks each and used in an undergraduate course. We use xv6 as a platform for these modules. Among the modules created are the authentication, access control, and address space layout randomization (ASLR) module. We have created our own implementations for each module. We compared the details and functionalities of our implementations with production grade OSs, such as Linux. The behavior of our implementations has been found to be relatively similar to that of systems such as Linux, although they differ in the depth of execution. At the time of writing, these modules have not been deployed for use in a real course, undergraduate or otherwise. We have plan to evaluate these modules in an undergraduate level OS course in Fall'22 and Spring'23. Various details will emerge from practical use and experience. Our reference implementations also have a few limitations that will be released in the future updates of the modules along with the students' feedback.

References

1. Ralph Langner, "Stuxnet: Dissecting a cyberwarfare weapon." IEEE Security & Privacy Vol. 9, No. 3, pp 49-51, 2011.
2. Colonial Pipeline ransomware attack, Available online: https://en.wikipedia.org/wiki/Colonial_Pipeline_ransomware_attack, Accessed Feb 15, 2022.
3. Lee Mathews, Florida Water Plant Hackers Exploited Old Software and Poor Password Habits, Available online: https://www.forbes.com/sites/leemathews/2021/02/15/florida-water-plant-hackers-exploited-old-software-and-poor-password-habits/?sh=188bc59e334e, Accessed Feb 15, 2022.
4. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, 1st ed. Arpaci-Dusseau Books, August 2018 [Online]. Available: www.ostep.org, Accessed Feb 13, 2022.
5. R. Cox, F. Kaashoek, and R. Morris. (2020, Aug.) xv6: a simple, Unix-like teaching operating system. Massachusetts Institute of Technology. Available: https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf, Last Accessed: Feb 13, 2022.
6. xv6-riscv, Available online: https://github.com/mit-pdos/xv6-riscv, Accessed Feb 15, 2022.
7. Andrew S. Tanenbaum, Minix3 (Online). Available: https://www.minix3.org/, Accessed Feb 13, 2022.
8. D. Comer, Xinu. [Online]. Available: https://xinu.cs.purdue.edu/#textbook, Accessed Feb 13, 2022.

9. Sixth Edition Unix source code, Version 6. [Online]. Available: https://minnie.tuhs.org/cgi-bin/utree.pl?file=V6, Accessed Feb 13, 2022.
10. J. Thyme, (2004, Jun.) Minix on Qemu Step-by-step. [Online]. Available: https://minix1.woodhull.com/faq/qemumx.html, Accessed Feb 13, 2022.
11. M. Spivey, Installing Minix 2 on VirtualBox. University of Oxford. [Online]. Available: https://spivey.oriel.ox.ac.uk/corner/Installing_Minix_2_on_VirtualBox, Accessed Feb 13, 2022.
12. D. Given, Minix QD [Online]. Available: https://github.com/davidgiven/minix2, Accessed Feb 13, 2022.
13. A. Prakash, Linux File Permissions and Ownership Explained with Examples [Online]. Available: https://linuxhandbook.com/linux-file-permissions/, Accessed Feb 13, 2022.
14. PaX team, ASLR [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt, Accessed Feb 13, 2022.
15. Stein, ASLR [Online]. Available: https://isopenbsdsecu.re/mitigations/aslr/, Accessed Feb 13, 2022.
16. A. van de Ven, Patch 0/6 virtual address space randomization [Online]. Available: https://lwn.net/Articles/120966/, Accessed Feb 13, 2022.
17. H. Marco-Gisbert and I. Ripoll Ripoll, "Address Space Layout Randomization Next Generation," MDPI Applied Sciences, vol. 9, no. 14, 2019.
18. P. Gonarkar, D. Arole, and P. Gondachwar, "Pedagogical tools for system software and operating system courses using xv6 kernel," B.Tech. Comp. Eng. project, College of Engineering Pune, May 2014. Available: http://foss.coep.org.in/fosslab/projects/xv6_new_assignments _project.pdf. Accessed Feb 13, 2022.
19. F. Kaashoek and R. Morris. (2020) 6.S081 / Fall 2020. Course schedule for 6.S081. Available: https://pdos.csail.mit.edu/6.828/2020/schedule.html, Accessed Feb 13, 2022.
20. X. Wang, L. Nelson, and N. Durand. (2019) Labs - CSEP 551. Laboratory assignments for CSEP 551. Available: https://courses.cs.washington.edu/courses/csep551/19au/labs/, Accessed Feb 13, 2022.
21. D. Mishra, "Gemos: Bridging the gap between architecture and operating system in computer system education," in Proceedings of the Workshop on Computer Architecture Education, WCAE'19. New York, NY, USA, 2019.
22. C. T. K. Koster, J. Wiglz, D. Wrecker, and B. S. B. Matthews "Authorization for xv6 OS Project," 2015. Available: https://github.com/CKost/Authorization, Accessed Feb 13, 2022.
23. J. Bui and N. Prasad. xv6 ASLR Project. Available: https://github.com/TypingKoala/xv6-riscv-aslr, Accessed Feb 13, 2022.
24. Johnmwu. xv6-aslr. Available: https://github.com/johnmwu/xv6-aslr, Accessed Feb 13, 2022.