Networks and Systems Security II - Winter 2025

Sambuddho Chakravarty

January 21, 2025

Assignment 1 (Total points: 140)

Requirements: A *nix VM on which you would create the following programs. Best is to create a *nix VM with a minimal installation (*i.e.*, no `xserver` (`Xwindow`) installed) such that memory and system requirements are the least. You could try Linux distributions that are suitable for text mode installation that use minimal resource and still use an old-school `init`, *e.g.* Artix Linux (*i.e.*, Arch without `systemd`), Devuan (*i.e.*, Debian without `systemd`), Alpine Linux, Slackware Linux, or several others that you may find here. Alternatively, you could work with a non-Linux Unix-like OS like FreeBSD, OpenBSD, NetBSD, DragonflyBSD.

Due date: Feb 10, 2025. Time: 23:59 Hrs.

1 Simple `sudo` (total points: 50)

This first assignment familiarizes you the `setuid()` system call through your effort of authoring your own `sudo` program. The program needs to be `setuid` program. `Setuid` programs have the `setuid` bits set. The owner your `sudo` program has to be `root`, along with the `setuid` and `execute` permissions enabled.

When your `sudo` program (call it `my_sudo`) it runs a program whose owner is `root` it would, run as `root`. However, when running a program whose owner is not root, you would need to switch to the non-root user, using system call `seteuid()`. You need to check for every corner case with regards to the functionality. Feel free to consider other possible assumptions. DO NOT forget to list the assumptions in the system description that you would submit.

Note: This assignment assumes real users (who have real UIDs), with real directory structures. You are supposed to write real programs, corresponding the aforementioned commands, just like you have in a real *nix system.

Grading Rubric

- Successful compilation using Makefile – 5 points.
- Correct functionality of your `sudo` program points, while handling all possible corner cases, *e.g.* accessing programs for which one has no execute permissions etc. — 20 points.
- Use of system calls like – getuid(), setuid() and seteuid() to implement the aforementioned functionality – 20 points.

- Successfully defending against atleast 2 attacks/bugs/errors/corner cases — 10 points (List the bugs/errors/attacks that you defend against).
- Readme with description of the systems, commands to execute and test the program and the assumptions that you made – 5 points.

2 ACL Shell (total points: 90)

For the second task, you need to create your own ACLShell much like `bash`, `ksh`, `tcsh`, `zsh`, *etc.*. First you create a directory, lets call it `my_acl_directory` in your home directory. The directory should have some dummy files and directories in it. Then you would need to use your existing laptop and desktop and create multiple *REAL* users -– lets call them 'narender', 'donald', 'vladimir', 'shahbaz', 'benjamin', 'amit" (whatever you feel like), *etc.* You would add these using the actual `adduser` command. The /etc/passwd file should reflect these users and their home directories. The files and directories in the dummy directory must belong to these users. You can do that by running the `chown` command as 'root' to change their ownerships.

Next you log in as one of these users, say 'amit'. From 'amit"s home directory run your `ACLShell` that shows you a prompt. In this shell, if type regular shell commands (*e.g.*, let us assume you are using `bash` shell commands), the command should be passed as argument to the directory `/usr/bin/bash` or `/usr/local/bin/bash`, whichever is correct path for the `bash` shell. However, for specific other types of commands, which are described next, they are not passed on to `bash` but handled separately.

In your dummy directory (mentioned above), the files you have, must have a simple set of ACLs present there in the file's extended attributes (supported in most *nix though with different system call signatures to manipulate them). These ACLs should be of the form of 'user':'name1,value2 name2,value2' pairs for allowing and denying the the access. *E.g.* it could be in the form of 'user':'narender,rw- vladimir,r– '. For simplicity you would only need to implement ACLs for usernames (not need to implement for groups). When you create these files and directories, for file you must set these ACL strings to the files attributes using commands like `setfattr` and `setextattr`, depending on your *nix installation.

You also need to implement individual programs for `fput`, `my_ls`, `fget`, `my_cd` and `create_dir`. The owner of each of these commands *must* be 'root' and their setuid bit must be enabled (using `chmod` command).

From your shell, that you run after logging in as one of the users you created, you try the following:

1. Try to read a file using `fget` or `my_ls` (if it is a directory *i.e.*). These commands must check the real caller user (using `getuid()` system calls to determine who he actual user is. Thereafter, using the target files or directories extended attributes (obtained through the *nix specific system call), you should decide to allow or deny reading the file. When reading the contents of the file, the effective user should drop down to the owner of the file or directory that is being read.
2. Similar to (1) above, `fput` (or `create_dir`) command must check if the caller user can write to a file (for sake of simplicity it could merely be an append) or create a directory (using the `mkdir()`) system call. These commands must check if the caller user can write to the target by fetching the ACL strings in the extended attributes and thereafter decide to allow or deny.
3. Similar to (1) and (2), the caller should be able to run a target program or access a target directory (by running the `my_cd` command). The task to run a target program must be mediated by the `ACLShell` itself that must check the target files ACLs based on the extended attributes. If the ACLs allow then the

shell must do a `fork()` and `exec()` to run the target program else must throw an error and stop.

4. If ACLs don't allow the requested permission, the called program (*i.e.*, programs you wrote like `fput`, `my_ls`, `my_cd` *etc.*) must check the target file's or directory's DAC permissions using the `access()` system call. This become effectively the default behavior in case ACLs deny access.

Access to the ACL would be via programs called `setacl` and `getacl`. These would be individual binaries with their `setuid` bit turned on and their owners would be 'root'. Whenever these programs are invoked they determine the actual user ID of the invoking program using `getuid()` system call. This is especially essential to ensure that only the owner should be in a position to change the ACLs of the target file and other users shouldn't. For `getacl` any user who is read permission on the parent directory *i.e* `my_acl_directory` in case of file inside it or for subdirectories, for files in the appropriate subdirectories, should be able to see the ACL printed on the terminal.

Grading Rubric

- Successful compilation using Makefile – 5 points.
- Use of system calls like – setuid() to implement the aforementioned functionality, along with the system calls used previously like readdir(), opendir(), read(), write() etc. – 30 points.
- Correct functionality of each of the programs demonstrated with appropriate test cases and examples – 7 programs x 5 points per correct functionality = 35 points.
- Successfully defending against atleast 3 attacks/bugs/errors -– 15 points (List the bugs/errors/attacks that you defend against).
- Readme with description of the systems, commands to execute and test the program and the assumptions that you made – 5 points.

Late Submission Policy

- Submitted on or before Feb. 14, 2025 (23:59 hrs) – No points deducted.
- Submitted after Feb. 14, 2025 but on or before Feb. 16, 2025 (23:59 hrs) – 5 points deducted.
- Submitted after Feb. 16, 2025 but on or before Feb. 18, 2025 – 10 points deducted.
- Submitted after Feb. 18, 2025 – no points shall be awarded.