Networks and Systems Security, Assignment 1

Due date: Feb 15, 2019
Total points: 40

Part I: ACLs and Setuid (Total points 40)

The objective of this assignment is to familiarize you with using setuid() (and related) system call(s). In this assignment you would need create multiple **REAL** users – say call them 'larry', 'sarah', 'bill', 'steve', 'mukesh', 'nirav', 'azim, 'mark' *etc*. (whatever you feel like). You would add these using the actual 'adduser' command. Their home directories should be inside the simple_slash directory that you created earlier. The directories could have names like simple_slash/home/larry (for e.g. for user larry). You need to accordingly modify the /etc/passwd file to indicate the home directories for each of these users. The directories 'simple_slash' and 'simple_slash/home' should have 'root' as the owner. You could additionally create a group called 'simple_slash' and the directory 'simple_slash' and 'simple_slash/home' could have their groups set to 'simple_slash'. The files and directories in each of the users' directories should be owned by the users themselves (and not by 'root'). *E.g.* simple_slash/home/larry should have the owner 'larry'. Additionally you would also require a new user called 'fakeroot' which would be have very similar to the actual 'root' user. The purpose of this would be clear ahead in the description.

You need to implement ACLs which should **override** DAC. These ACLs would be stored with the files. One way to do this is to use a C structure / class that has an array of strings representing ACLs, along with a pointer to a character buffer which can be modified based on the number of ACL entries in the file. *E.g.*

```
struct file_acl_data{
unsigned char **acl; // ACL strings
unsigned int acl_len; // Number of acl strings
} ;
```

Access to the ACL would be via external programs called 'setacl' and 'getacl'. You need to implement your own semantics for ACLs to prioritize it over DAC or vice versa. You need to implement your own 'setacl' and 'getacl' **programs** which would be setuid program (having the setuid bit set). These programs would be used for manipulating the file ACL entries. The owner of the program would be the user 'fakeroot'. The program starts by checking if you have privileges to manipulate the entries or not. If you do then you could actually modify the ACL entries for the file. Normally, only the program owner and 'fakeroot' can modify the entries.

Further, you need to create your following programs as well (other than `setacl' and `getacl'):

'ls': Lists file attributes, ownerships, permissions, sizes etc.

'fput' / 'fget': The behaviour of this programs is same as that of the one in the previous assignment, i.e. used to edit a file.

'create_dir': The behaviour of this program is same as that used in the previous assignment, i.e. used to create a directory.

‘do_exec’: This is very similar to the ‘sudo’ program and is used to run programs owned by other users, while acquiring privileges of the owner. For e.g. if user `sarah’ wants to run programs owned by user ‘mukesh’ she would run it by executing the command – do_exec simple_slash/home/mukesh/prog1, assuming prog1 is owned by user ‘mukesh’.

But, these programs would be setuid programs and owned by ‘fakeroot’. Each time any user invokes these programs, the programs identify the calling user AND the ACLs stored in the accessed files/programs.

Big question: *Where do you store the ACLs?*

You would need to use the Linux extended file attributes filed to store the ACL as a file metadata. You may access the extended file attributes using the system calls setxattr() and getxattr().

Whatever semantics you may choose, you would need to document them. Only a file/directory owner should be able to modify the ACLs associated to its file or directory.

The programs must also print the actual userID/username of the effective user immediately before and after accessing the file or directory (just to make sure that the programs are working correctly). You would need to use the *setuid()* family of functions to achieve this functionality.

You need to check for every corner case with regards to the functionality of access controls.
Feel free to consider all possible assumptions. DO NOT forget to list the assumptions in the system description that you are required to deliver.

What you are supposed to submit:
1. C source code for the aforementioned shell server and client programs.
2. Makefile through which one could compile these programs.
3. A write up of what your system does, what all assumptions you made, the inputs that you used to test your program and all the errors that you handled, the Threat Model that you considered and how your system defends against those.

How you would be graded:
1. Successful compilation using Makefile – 5 points
2. Use of system calls like – setuid() to implement the aforementioned functionality, along with the system calls used previously like readdir(), opendir(), read(), write() etc. – 10 points.
3. Successfully handling various access control scenarios – e.g. users attempting to access a file which the DAC doesn’t allow by the ACLs allow or vice versa -- 10 points
4. Successfully defending against atleast 3 attacks/bugs/errors – 10 points (List the bugs/errors/attacks that you defend against)
5. Description of the systems, commands to execute and test the program and the assumptions that you made. – 5 points

Part II: Netfilter Kernel Module (Bonus Points Assignment, total points: 40)

Description:

This assignment would require you to write a Linux kernel module that would involve using the netfilter framework (like what was described in the class). You would require to test it over the VMs created for lab exercise 1. The VM1 communicates to VM3, via VM2. The VM2 acts as the router/firewall. On VM1 you would now require installing the network reconnaissance tool called `nmap' (URL provided ahead). Using `nmap' you could run various kinds of network reconnaissance exercises on the virtual machine VM3.

You would require to write a netfilter kernel module (on VM2) that intercept packets and log them in the kernel logs. Your module should be able to detect ***any four*** different kinds of TCP based reconnaissance packets that can be generated using 'nmap'. The module should detect these packets them to kernel logs (syslog). You do not have to filter the packets, merely identify the kind of network reconnaissance and log them.

Some useful resources:

https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html

https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture

http://www.paulkiddie.com/2009/10/creating-a-simple-hello-world-netfilter-module/

http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction

https://www-users.cs.york.ac.uk/~jac/PublishedPapers/NetworkReconnaisance.pdf

https://www.computernetworkingnotes.com/ccna-study-guide/network-security-threat-and-solutions.html

https://nmap.org/book/man.html

What you are supposed to submit:

1. C source code for the aforementioned shell server and client programs.
2. Makefile through which one could compile these programs.
3. A write up of what your system does, what all assumptions you made, the inputs that you used to test your program and all the errors that you handled.

How you would be graded:

1. Successful compilation using Makefile – 10 points.
2. Demonstration of detection of the TCP reconnaissance packets – 25 points.
3. Description of how you are detecting the packets etc. – 5 points.

LATE SUBMISSION POLICY:

- Submitted on or before February 16, 2019 – No points deducted.
- Submitted after February 16, 2019 but on or before February 18, 2019 – 5 points deducted.
- Submitted after February 18, 2019 – 10 points deducted.
- Submitted after February 20, 2019 – No points awarded.