02/16

System Security Class Notes

(Linux lpr command, System vs. Language Capabilities, Revocation)

Aravinda Kidambi Srinivasan

Lpr command

- lpr is the linux print command.
- It needs root privileges to write to the printer queue.

Linux implemented it by giving *setuid privilege,* which ideally lets any user to run this program as a root.

```
$ ls -l lpr
rwsrwxr-x   root     root     lpr
```

the 's' bit in the permission suggests anyone running lpr now get root permissions to run the command.

lpr.c

```
1. pfd = open("/dev/printer",w);
   ...
2. tfd = open(argv[3],r);
```

Lpr always runs with root privileges. So it has privileges to open the printer file

Opens the file to print. However it uses to the root privileges to open this file too and always succeeds

Lpr has ambient authority and this could be easily exploited; anyone could just print the password file. But the intention was

- To use the root permission at line 1.
- And to use the users permission at line 2

Fix 1 – start with root hat

1. Open printer file
   ...
2. Switch_hat(user)
3. Open file to be printed
4. Switch_hat(root)

- This violates 'Least privileges' mechanism

Fix 2 – start with user hat

1. Open file to be printed
   ...
2. Switch_hat(root)
3. Open printer file
4. Switch_hat(user)

- Employs *least privileges*
- But. *Easy to make mistakes*. It could leave the program in root hat if the programmer forgot the switch back to user hat.
- Can lead to **Tractor beaming** attack

Proposal

- Give print privilege to all users
- Provide access to print program based on this privilege
- The program doesn't need root permissions
- *Problem:* can lead to DOS attack – an attacker watching a user issuing a print command, to immediately print a different file to overwrite the printer queue.
- More refined access control mechanism to allow only append to the printer queue is required

Fix3 - capability based approach

- Make the print program more modular.
- Keep code to open the file to be printed outside the lpr program
- $lpr –p printer < myfile
- Shell opens the file 'myfile' using users permission and passes as input to the print program
- Print program can have root access to open printer but an attacker cannot misuse this privilege to open a protected file.

Fix 2 vs. Fix 3

Fix2

- An untrusted program employing fix2 has root access
- a malicious programmer could have used this privilege to delete the entire file system
- This doesn't solve the mutually distrusting users problem

Fix3

- There is no switching of hat involved.
- It runs with no file-opening capability
- Hence a user can still safely run an untrusted program employing fix3.
- Solves the mutually distrusting users problem

Capabilities

- Is an unforgeable handle on an object that grants the process to perform some operation on the object
- Eg. Unix file descriptor
  - Is an index into open file descriptor table
  - Is a number between, say 1 and 1000, can it ensure unforgeability ?
    - It's an index within the process' file descriptor table.
    - A process cannot modify this table maintained by the kernel apart from using the 'open' system call
    - Therefore, what a FD index refers to cannot be forged
  - Entries in table specify (a capability)
    - The file that can be access via the FD
    - Allowed operations

- Every program starts with
  - Stdin, stdout and stderr – standard file descriptors
  - No process can fake a file descriptor for a file it doesn't have permissions on

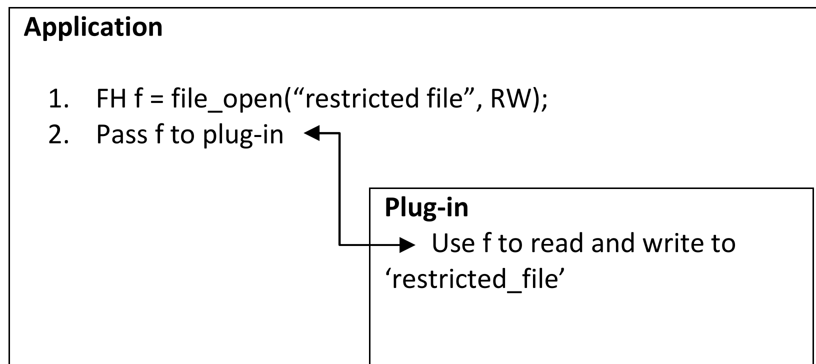
Pure Capability World

- Program starts with
  - Stdin, stderr, stdout
  - Any other FH(file handle) to any file/directory it is allowed to access

1. To run a program on one file
   a. Open target file
   b. Launch program and pass the FH to it
   c. This controls what the program can access.
   d. No access violations can occur
2. To give access to a directory
   a. Open dir for RW
   b. Launch a program passing the capability(FH) for dir
   c. Confine: no capability to *traverse* ".."

System vs. Language Based Capabilities

- Strongly typed languages e.g. Java, have unforgeable pointers
- Think of a plug-in for an application; both running in the same address space. Using C-pointers, a piece of plug-in code can access the entire memory.
- In Java, objects are used to provide restricted access ≡ Capabilities



How to pass f to plug-in with only read permissions?

- Fix 1
  - Use the Object Hierarchy in Java.
  - Object -→ … ReadOnlyFile → File
  - Use type casting on line 2, to pass ReadOnlyFile(f) to the plug-in
  - Problems: Java allows down casting; plug-in can use File(f) to still write to 'restricted_file'
- Fix 2

- Create a wrapper around File class
- Class ReadOnlyFile {
    - Private File f;
    - ReadOnlyFile(File f) {
        - this.f = f;
    - }
    - Read() {
        - Read from FH f
    - }
    - Write() {
        - Throw exception("Write Access not allowed");
    - }
- }
- Problems: Once object is passed to plug-in by
    - Pass roFH = new ReadOnlyFile(f)
    - It cannot be revoked. Plug-in gets a persistent read access on 'restricted_file'

- Fix 3
    - Create a revocable wrapper class
    - Class RevocableROFile {
        - Private File f;
        - Private boolean revoked=false;
        - ReadOnlyFile(File f) {
            - this.f = f;
        - }
        - Read() {
            - If not revoked
                - Read from FH f
            - Else
                - Throw exception("Read Access revoked");
        - }
        - Write() {
            - Throw exception("Write Access not allowed");
        - }
        - Revoke() {
            - Revoked = true;
        - }
    - }
    - Line 2 becomes rf = new RevocableROFile(f); and pass rf
    - If Plug-in misbehaved, Application can always call rf.revoke();
    - Problems: Plug-in can

- Clone object before revocation. Can still get access to ‘restricted_file’
- Serialize the object when it receives. Can de-serialize and get access to ‘restricted_file’ even after its original object was revoked.
- Solution: Override clone() and serialize() and throw exception.