File Systems – Practical Avoidance of Security Vulnerabilities

Administrivia and Questions

- Oral midterm – much like job interviews with some open-ended questions (as we've practiced in class, sections, homework, and projects all quarter long). If after taking this class your ability to reason in open-ended situations has improved (whether in scenarios concerning OS or not), we will consider our job well done).

File System Issues

- We have said in lecture that access checking on file operations is performed on file open
  `int fd = open("/tmp/myfile", "r");`
  and not on subsequent operations in order to save the overhead of verifying that access is allowed if the necessary check has already been done once before. But is it really so simple?
  - What if one attempts to write to a file that was open for reading only? When would the access be denied – immediately or at some point in the future? It turns out that there are situations in which both approaches make sense:
    - Immediately
      - If a programmatic write operation is attempted (e.g., `write(fd, buf, 10)`), it is disallowed immediately. The per-process open file table (PPOFT) stores references to all open files for the given process, in addition to the access rights these files were open with. These are checked on every operation.
    - Sometime later
      - One can be allowed to write to a buffer and only when an attempt is made to save the contents of the buffer to a file will it become clear that this operation is not allowed. This is the approach taken by text editors on UNIX (and perhaps other operating systems) - when a user attempts to open a file (e.g., "`pico myfile`"), then don't specify (for simplicity?) if they would write to it or not. If the file is read-only, writes will still be allowed (but to a buffered copy of the file) and later disallowed on exit and/or save. This has the potential to waste a lot of user time and effort.

- Do we have to refer to a file using its file descriptor? Can we not use the filename instead? It turns out that this would lead to a race condition of the type TOCTTOU (time-of-check to time-of-use). Consider this:
  - The OS kernel needs to copy the parameters of a system call before checking them.
    - Why? Otherwise they will be under user control and might be (erroneously or maliciously) changed after the kernel verifies them, but before it actually uses them. For instance, if a pointer was passed in that initially pointed to user space (and hence verified correctly) but at some point later was changed to point to kernel space, the OS will be tricked into dereferencing its own memory and exposing the result to the user.
  - A similar strategy needs to be employed in the case of file access to make it secure. Do you see the vulnerability in the following code? (Assume that `access()` and `openfile()` are user routines.)

```
if ( !access("myfile", WRITE_OK) ) {    /* 1 */
   f = openfile("myfile", "wb");        /* 2 */
   write(f);
}
else {
   perror("Access denied!");
}
```

- With the right timing, a user (attacker) may replace the file `myfile` (for which the user has proper permissions) with one of its choosing (to which it does not have permissions) without the process executing the code above realizing that a change has happened. In the code above line 1 is the time of check while line 2 is the time of use. If what is represented by `myfile` changes between line 1 and line 2, a vulnerability results since access has been granted in line 1 but *not* to what `myfile` refers to now. An attacker may exploit it by forcing the following

  ```
  rm myfile; ln –s /etc/passwd myfile
  ```

  to execute between the execution of line 1 and line 2. (With sufficiently many attempts on a local machine, this could be made to happen.) In our case, `/etc/passwd` (the password file on UNIX systems) is simply an example of a resource to which a normal user does not have write access, but to which a process running with *root* (highest) privileges does have access.
- What are the necessary conditions for the success of the above exploit? Knowing them is a first step toward devising ways to prevent it (or at least decrease the chances for it happening).
  - Attacker must (generally) have local access to the machine where the above code is running.
  - Attacker must be able to create a symbolic link in the protected directory.
  - Exploited code must run with high privileges (higher than those of the attacker).
- Avoiding TOCTTOU vulnerabilities
  - Be aware that file system routines that take a filename (a string) as parameter are vulnerable. Whenever possible, use routines that take file descriptors (handles) instead. By itself, this measure would decrease the risk, but not eliminate it, since there are routines that only take a filename: e.g., `unlink()`, `mkdir()`, etc. For suggestions on how to effectively deal with this, look at the suggested resource below (p.222).
  - Systems in which applications run with the minimal necessary privileges are more robust than systems with only a few privilege levels. The latter use *root* (in UNIX terminology) or some other master account for applications that need to access some protected resources, but in doing so these systems expose access to all protected resources. All an attacker needs is an exploitable vulnerability somewhere in these (setuid[1]) applications. Commercially available systems don't follow this strategy, so the feasibility of this approach is mostly a research topic.
  - Verify that the file that was intended for opening is actually the one that was opened:
    (1) `lstat()` and save the `lstat_info`;
    (2) `open()` given the filename;
    (3) `fstat()` on the file descriptor returned from `open()` and save the `fstat_info`;
    (4) compare `lstat_info` to `fstat_info` – the relevant fields in them should be exactly the same if the file was open securely.

    This approach is the most reliable and practical known means of ensuring secure file access.

Resources used: "Building Secure Software", http://www.buildingsecuresoftware.com/

[1] Most applications run with the privileges of the account of the user that started them. For those EUID (effective user ID) is the same as UID (user ID) of the user. Applications can however be explicitly marked to be *setuid*, which means that their EUID becomes the same as the UID of the owner of the application (typically root or some high-privileged account). This is mostly to allow the application access to resources that the user herself does not have direct access to. An example use of *setuid* is for logging user activity: typical users shouldn't have write access to the log file (lest they decide to tamper with it), yet the applications they run should have write access to that file.