

# Intro to Cybersecurity – Spring 2023

## Homework/Lab #2

Due: Monday, May 10th 2023

### Highlights

- Expected contribution towards the final score: 6%.
- **You should work on this homework individually or in teams of up to 5 members (highly recommended). One submission per team.**
- **Submit your work in PDF as a group** through the USTC Blackboard
  - Choose one group named as “Homework/Lab #2 [1-100]” on Blackboard (工具 -> 小组), and submit as the group, which makes it easier for the teaching team to register your score.

- 1) **(20 points)** In Unix, every process has a real user id (ruid), an effective user id (euid), and a saved user id (suid). Processes with an euid of 0 have special root privileges.  
Hints: Read the background components (Sec 3, 4, and 5.1) of [this research paper](#).
- a) **(2 points)** If a process with user id  $x$  forks to create another process, what user id does the new process have? (Hint: it's the same answer for euid, ruid, and suid.)
- b) **(6 points)** If a process with euid  $y$  makes a `setuid` system call, what possible euids can the process run with after the call, in each of the following situations:
- Before:  $\text{euid} = y > 0$ , saved user id  $\text{suid} = m$  and real user id  $\text{ruid} = m$ . After:?
  - Before:  $y = 0$  After:?
- c) **(3 points)** Each Android application runs in a separate process using a separate user id. From a security standpoint, what is the advantage of assigning separate uids instead of using the same uid for all? Explain.
- d) **(4 points)** The Android zygote process that creates new processes runs as root. After forking to create a new process, `setuid` is normally called. Explain what uid the new process has initially and why it is important to call `setuid`? What security purpose does this serve?
- e) **(5 points)** When a Unix user wishes to change her password, she uses the `passwd` program. The Unix password file is usually publicly readable but (for obvious reasons) can only be written by processes with root privileges.
- How should the `setuid` bit be set on this `passwd` program? Explain how this lets a user change her password.
  - Why does this make it important to write the `passwd` program source code carefully?

### Solutions:

- a)  $x$
- b) When euid is not 0, only suid and ruid can be set, in this case, it is  $m$  or  $y$ ; when euid is 0, the resulting euid can be any
- c) It provides isolation among applications, so that one application cannot access directories and files owned by another app (with a different user id) by default.
- d) The new process has the root uid initially, and that's why it is important to call `setuid` and set up the effective UID as the non-privileged one associated with the specific app. Similar to c), it is designed to isolate applications and prevent an application from running as root
- e) The `setuid` flag should be set as 1, and when a user calls this program, the effective user id of the forked process will be set as root, so that the forked `passwd` program can modify the password file and thus change password. Besides, `passwd` must be written carefully because it should guarantee a user except for the root can only change password for herself.

- 2) **(5 points)** Assume that passwords are limited to the use of the 95 printable ASCII characters and that all passwords are 10 characters in length. Assume a password cracker with an encryption rate of 6.4 million encryptions per second. How long will it take to test exhaustively all possible passwords on a UNIX system?

**Solutions:**

There are  $95^{10} \approx 6 \times 10^{19}$  possible passwords. The time required is:

$$\frac{6 \times 10^{19} \text{ passwords}}{6.4 \times 10^6 \text{ passwords/second}} = 9.4 \times 10^{12} \text{ seconds} \\ = 300,000 \text{ years}$$

- 3) **(15 points)** Consider the following code snippet:

```
if (!stat("./file.dat", buf)) return; // abort if file exists
sleep(10);                          // sleep for 10 seconds
fp = fopen("./file.dat", "w");       // open file for write
fprintf(fp, "Hello world" );
close(fp);
```

- a) **(5 points)** Suppose this code is running as a setuid root program. Give an example of how this code can lead to unexpected behavior that could cause a security problem. Hint: try using symbolic links.
- b) **(5 points)** Suppose the sleep(10) is removed from the code above. Could the problem you identified in part (a) still occur? Please explain.
- c) **(5 points)** How would you fix the code to prevent the problem from part (a)?

**Solutions:**

- a) *It leads to a TOCTOU (time-of-check, time-of-use) race condition. During the process sleep, a symbolic link named file.dat can be created and be linked to a security-sensitive file, e.g., password.*
- b) *Yes, it can still occur, despite being with a lower probability. Once the existence check passed, the CPU may be switched to another process rather than continuing to run the next line of code, i.e., the file open operation*
- c) *[This article](#) provides more detailed discussion on TOCTOU, as well as how to fix this issue/*
- 4) **(10 points)** It was stated that the inclusion of the salt in the UNIX password scheme increases the difficulty of guessing by a factor of 4096. But the salt is stored in plaintext in the same entry as the corresponding ciphertext password. Therefore, those two characters are known to the attacker and need not be guessed.
- a) **(5 points)** Why is it asserted that salt increases security?
- b) **(5 points)** Wouldn't it be possible to completely thwart all password crackers by dramatically increasing the salt size to, say, 24 or 48 bits?

**Solutions:**

- a) *The salt can help mitigate the dictionary attack wherein the attacker has access to a password dictionary as well as a stolen password file*
- b) *the salt size will only have an impact on the possibilities of two password entries sharing*

*the same salt. It has little impact on thwarting password crackers. In the scenario where the attacker has access to a password file of  $n$  entries, and a dictionary of  $m$  entries. It will do  $n * m$  hash calculations to crack all these entries. Also, a user may set up a very simple password that can be easily figured out by the attacker.*

- 5) **(10 points)** The VAX/VMS operating system makes use of four processor access modes to facilitate the protection and sharing of system resources among processes. The access mode determines:

- **Instruction execution privileges:** What instructions the processor may execute
- **Memory access privileges:** Which locations in virtual memory the current instruction may access

The four modes are as follows:

- **Kernel:** Executes the kernel of the VMS operating system, which includes memory management, interrupt handling, and I/O operations
- **Executive:** Executes many of the operating system service calls, including file and record (disk and tape) management routines
- **Supervisor:** Executes other operating system services, such as responses to user commands
- **User:** Executes user programs, plus utilities such as compilers, editors, linkers, and debuggers

A process executing in a less-privileged mode often needs to call a procedure that executes in a more-privileged mode; for example, a user program requires an operating system service. This call is achieved by using a change-mode (CHM) instruction, which causes an interrupt that transfers control to a routine at the new access mode. A return is made by executing the REI (return from exception or interrupt) instruction.

- a) **(5 points)** A number of operating systems have two modes, kernel and user. What are the advantages and disadvantages of providing four modes instead of two?
- b) **(5 points)** Can you make a case for even more than four modes?

#### **Solutions:**

- a) *The advantage of four modes is that there is more flexibility to control access to memory, allowing finer tuning of memory protection. The disadvantage is complexity and processing overhead. For example, procedures running at each of the access modes require separate stacks with appropriate accessibility.*
- b) *In principle, the more modes, the more flexibility, but it seems difficult to justify going beyond four.*

#### **6) (20 points) A lab to understand user/password management on Unix/Linux**

- a) **(4 points)** Create a user X with home directory, and set up its password Y (hints: commands `useradd` and `passwd`)
- b) **(2 points)** Look into the `passwd` file (`/etc/passwd`), and locate the entry of your newly created user X, Look into the file (`/etc/shadow`) storing the salted password hash, identify the entry for your newly created user X
- c) **(5 points)** Understand the shadow entry format, parse out the salt, the salted password hash, as well as the hash algorithm
- d) **(5 points)** Utilize `openssl passwd` to recalculate the password hash, and compare with the one stored in `/etc/shadow`
- e) **(4 points)** Change the password for User X, and redo d

**Solutions:**

- [The entry format of /etc/shadow](#)
- [openssl passwd](#)

- 7) **(20 points)** Select and read one of the following papers, summarize its ideas, and give your critical reviews:
- a) Backes, Michael, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. **"Android security framework: Extensible multi-layered access control on android."** In Proceedings of the 30th annual computer security applications conference, pp. 46-55. 2014.
  - b) Barth, Adam, Collin Jackson, Charles Reis, and TGC Team. **"The security architecture of the chromium browser."** In Technical report. Stanford University, 2008.