CSci 4271W Developing Secure Software Systems (section 010)

Homework 3 **Due: Tuesday, March 4th, 2025**

Ground Rules. You may choose to complete these exercises in a group of up to three students. Each group should turn in **one** copy with the names of all group members on it. You may use any source you can find to help with this assignment but you **must** explicitly reference any source you use besides the lecture notes or assigned readings. No answers should come from people outside your group, or from AI tools like ChatGPT. If you use an AI tool to revise your writing, save a copy of the first draft you wrote yourself to provide it was originally your work. Electronically typeset copies of your solution should be submitted via Gradescope by 11:59pm on Tuesday, March 4th, 2025.

1. **More Defensive Programming.** Let's practice finding "bad programming practices" that could lead to exploits. Below are two buggy programs. For each program, (i) describe at least four bad things that could happen when running this problem in situations that the programmer probably didn't think of, including the programming mistake, the problematic situation, and the bad outcome, and (ii) Provide a safer implementation for this program (fragment). Note that your new implementation might have to behave differently than the old implementation in some cases; it is up to you to pick an interpretation and explain your rationale.

a. This code may run with elevated privilege, and has the user enter a message using a text editor that will be placed in a log file:

```
import os, os.path, tempfile
def updateLog(logFileName):
  editor = "nano"
  ued = os.getenv("EDITOR")
  if ued is not None and ued[:8] == "/usr/bin":
    editor = ued
  (fd,tmpfilename) = tempfile.mkstmp()
  os.close(fd)
  os.system("{} {}".format(editor, tmpfilename))
  with open(tmpfilename, "r") as f:
    msg = "\n".join(lines(f))
  with open(logFileName, "a") as l:
    l.write("Entry from " + os.getenv("USER") + "\n---\n")
    l.write(msg)
    l.write("---")
```

b. This code merges two files into columns of a single csv file:

```
void csvzip(char *src1, char *src2, char *dst) {
  char *dstline;
  char *sl[2];
  int i;
  FILE *sf[] = { fopen(src1, "r"), fopen(src2, "r") };
  FILE *df = fopen(dst, "w");
  while (!feof(sf[0]) && !feof(sf[1])) {
    for(i = 0; i < 2; i++) {
      sl[i] = malloc(LINE_MAX);
      fgets(sl[i],LINE_MAX,sf[i]);
    }
    sl[0][strlen(sl[0])-1]=',';
    dstline = malloc(strlen(sl[0])+strlen(sl[1])+1);
    for(i = 0; i < 2; i++) strcat(dstline,sl[i]);
    fprintf(df,"%s",dstline);
  }
}
```

2. **Passwords**. We don't usually think of them this way, but password-based authentication can also have "false negatives" like biometrics: "physical noise" can result in an incorrect reading of the password the user "knows", through typing mistakes. Suppose we were to catalog the most common password typing mistakes (maybe, for example, fingers off by a row, left-and-right hand letter swaps, missing a letter) and either have the login prompt try "undoing" each of these mistakes OR having the password file store hashes of each of these "mistake passwords". Presumably this would reduce the *insult rate* of our login system.

   a. How would this scheme affect other threats to authentication? Is the answer different when the errors are applied at hashing time vs at login time?

   b. How would the length of a user's password interact with the threats to authentication under this scheme? What tradeoffs might we make to minimize this impact?

3. **Access Control I**. A "Confused Deputy" bug happens when a program that needs access to a resource or file for one reason can be "manipulated" into accessing a resource or file inappropriately. For example, we might have a program that runs as root (using the setuid bit) that can be used to update the system login message given a text file:

```
#include <iostream>
#include <fstream>
using namespace std;

int print_usage() {
  cout << "set-motd: copy input file to login message" << endl;
  cout << "requires 1 argument" << endl;
  return 1;
}

int main(int argc, char **argv) {
  if (argc < 2) return print_usage();
  ofstream motd("/etc/motd");
  ifstream tocopy(argv[1]);
  while (tocopy) {
    motd.put(tocopy.get());
  }
  tocopy.close();
  motd.close();
  return 0;
}
```

   a. Why is this a confused deputy? (Hint: what might we use as an input file?)

   b. We could resolve this issue by using the `setuid` functions described in class and in chapter 3 of Wheeler. Describe how to modify the program to do so.

   c. Suppose we wanted to limit the set of users who could run this program to those belonging to the group `messagers`. What permissions settings should we set on the executable to accomplish this?

4. **Access Control II**. Login to your `csel-xsme-f24-csci4271-NNN` VM, and execute the following command:

```
$ find /bin /usr/bin -perm -u=s -exec ls -l \{\} \;
```

   The `find` command is a very useful unix tool for searching a directory tree for files matching a pattern; here we are looking for files that have the setuid bit on, and having `find` print the full directory listing

for each one. You'll see a list of several common programs that run as root. Choose 2 of these, and for each one, use `man` to find out what the program does. Describe why it might need root privileges. Then look at the manual page for capabilities (type `man capabilities`) and see if you can find a capability set we could assign the program so that it could do its job without full root privileges.