Universität des Saarlandes
Lehrstuhl für Softwaretechnik
Prof. Andreas Zeller, Stephan Neuhaus
Design of Secure Software Systems
Exercise 2

Total number of points obtainable: 33, plus 2 bonus points.

1 Password Aging

One popular technological means to guard against password problems is *password aging*. Password aging works by giving a password a lifetime. Once the password's lifetime is over, the user is forced to choose a new password. Usually, the system guards against the new password being the same as the old password.

- **(2+2 points)** Give at least two scenarios by which users can effectively avoid memorizing a completely new password each time their old password expires.
- **(4+4 points)** How would you implement a password aging system that does not allow users the loopholes that you identified in the previous exercise? Can you make the system usable *and* secure?
- **(2+2 points)** Against which threats does password aging help? Is the effort you expended in the previous exercise worth it?

Note that there are no right or wrong answers to these questions. However, your answer must still be well grounded.

2 Password Storage

Below is a program that is supposed to add users to a password database. There are several problems with this program. Try to find them simply by looking at the code, not by executing it. (This is the reason why the code isn't downloadable.) Some preliminary remarke:

- The program is a bit long, but it is really quite straightforward, so don't be intimidated by it.
- Security code reviews aren't for everyone. Before you try to find this program on the Internet, try to solve the questions on your own first. If you can't get a handle on it, ask your friendly TA or me next. If you still can't find anything wrong with the program, just admit defeat and see what the others have found out.
- The notation '*crypt*(3)' is traditionally used among Unix programmers to refer to the entry for 'crypt' in section 3 of the Unix manual. To get information on the *crypt*() function, you would type '`man 3 crypt`' in a shell.
- You can assume that the code compiles and runs OK, without errors or warnings, on at least some Linux systems.

Now, the questions:

- **(2 points, 2 bonus points)** The program's user is at some point asked to enter the password twice, to guard against misspellings. However, there's something wrong with that section of code. What is it? If you have some Unix programing knowledge, suggest a way to solve the problem.
- **(4 points)** The *crypt*(3) function is used in a dangerous manner. Why is that and how would you solve the problem?
- **(4 points)** The code has the user's plaintext password in memory for extended periods of time. Why is that a problem? (Note: You don't have to suggest a way to solve this problem yet.)
- **(4 points)** Assume that this program is supposed to be run by users other than root. What problems (if any) does this have and how could you solve them? (Warning: Difficult question; the solution to the problem needs advanced Unix programming knowledge.)

- **(3 points)** What happens if the portion of the program's data that contains the user's plaintext password is swapped out to disk? Assess the risk of this.

```
#include <crypt.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pwd.h>
#include <sys/types.h>

#define PW_FILE "/home/neuhaus/test.pw"
#define UNAME_INVALID 1
#define UNAME_EXISTS 2
#define NEW_USER_PROMPT "Enter a username: "
#define OUR_BUFSIZE 4096

FILE* pw_file;

static void open_password_file() {
  pw_file = fopen(PW_FILE, "r+");

  if (!pw_file) {
    fprintf(stderr, "Could not open %s for reading and writing.\n"
                "You must run 'touch %s' before this program "
                "will work.\n"
                "Also, make sure the current user has write "
                "permissions to the file.\n", PW_FILE, PW_FILE);
    exit(1);
  }
}

/* Read input up to a newline. */
static char* read_line(const char* prompt) {
  char* buf = (char*) malloc(sizeof(char) * OUR_BUFSIZE);
  char* res;
  int n;

  if (!buf) {
    fprintf(stderr, "Out of memory, exiting.\n");
    exit(1);
  }

  /* Print out the prompt. Must fflush to make sure it gets seen. */
  fprintf(stdout, "%s", prompt);
  fflush(stdout);

  /*
   * The algorithm reads the the standard input in chunks of OUR_BUFSIZE
   * characters. If the current chunk does not contain a newline character,
   * the buffer is prolonged by OUR_BUFSIZE more characters and the next
   * chunk is input.
   */
  res = fgets(buf, OUR_BUFSIZE, stdin);

  if (!res) {
    fprintf(stderr, "Error in reading input, exiting.\n");
    exit(1);
  }
  n = strlen(res);

  /* Read until end-of-line, re-allocating buffer as we go. */
  while (buf[n - 1] != '\n') {
    buf = (char*) realloc(buf, n + OUR_BUFSIZE);
```

```c
    if (!buf) {
      fprintf(stderr, "Out of memory, exiting.\n");
      exit(1);
    }
    res = fgets(buf + n, OUR_BUFSIZE, stdin);

    if (!res) {
      fprintf(stderr, "Error in reading input, exiting.\n");
      exit(1);
    }
    n += strlen(res);
  }
  /* Replace the newline with a null */
  buf[n - 1] = '\0';
  return buf;
}

/* Returns 0 if parameter is a valid user name. */
static int valid_user_name(const char *name) {
  int i;
  struct passwd* pwent;          /* See getpwent(3) */

  /* First check to make sure all the characters are valid. */
  for (i = 0; i < strlen(name); i++) {
    if (!isalnum(name[i]) && name[i] != '_') {
      return UNAME_INVALID;
    }
  }

  /* Now look up name in password file. */
  fseek(pw_file, 0, SEEK_SET);
  pwent = fgetpwent(pw_file);    /* See fgetpwent(3) */

  while (pwent) {
    if (!strcmp(name, pwent->pw_name))
      return UNAME_EXISTS;
    pwent = fgetpwent(pw_file);
  }

  return 0;
}

static char* get_new_user_name() {
  char* uname;
  int err;
  uname = read_line(NEW_USER_PROMPT);

  while ((err = valid_user_name(uname))) {

    switch (err) {
    case UNAME_INVALID:
      fprintf(stdout,
        "Invalid user name. Use letters, numbers and underscore only.\n");
      break;
    case UNAME_EXISTS:
      fprintf(stdout, "Username already exists.\n");
      break;
    default:
      fprintf(stdout, "Unknown error.\n");
      break;
    }

    free(uname);
    uname = read_line(NEW_USER_PROMPT);
  }

  return uname;
```

```c
}

static char* get_new_crypted_password() {
  char* pw1;
  char* pw2;
  char* crypted_pw;

  while (1) {
    pw1 = read_line("Enter password: ");
    pw2 = read_line("Reenter to confirm: ");

    if (strcmp(pw1, pw2)) {
      fprintf(stdout, "Passwords were not the same!\nTry"
              "again.\n");
      free(pw1);
      free(pw2);
      continue;
    }
    break;
  }

  crypted_pw = crypt(pw1, pw1);  /* See crypt(3) */
  free(pw1);
  free(pw2);
  return crypted_pw;
}

static void add_entry_to_database(const char *username, const char *password) {
  struct passwd pw;

  pw.pw_name = (char*) username; /* Discard const */
  pw.pw_passwd = (char*) password; /* Discard const */
  pw.pw_uid = 1;               /* Ignored in our application. */
  pw.pw_gid = 1;               /* Ignored in our application. */
  pw.pw_gecos = "";            /* Ignored in our application. */
  pw.pw_dir = "";              /* Ignored in our application. */
  pw.pw_shell = "";            /* Ignored in our application. */
  putpwent(&pw, pw_file);
}

int main() {
  char* username;
  char* password;

  open_password_file();
  username = get_new_user_name();
  password = get_new_crypted_password();
  add_entry_to_database(username, password);
  fprintf(stdout, "Added user %s...\n", username);

  return 0;
}
```