Homework 3: Secure Coding in C

See Webcourses and the syllabus for due dates.
This homework is secure coding practices.

General Directions

This homework is intended for individuals. You can do it in groups, but if you do so, be sure to follow the certify that you all understand and participated in the solutions as required by the course grading policy, see the cooperation section of the course grading policy.
You can use gcc with various flags and any other static analysis tools you like to help in your code reviews. If you use such a tool, let us know what tool you used, so that we can learn from your experience.

What to turn in

For problems with code answers, upload a text file with the appropriate suffix (like `.c`) to webcourses. The C files that accompany the problems are found in hw3.zip). Your task will be to review the code, find the parts of the code that enable various attacks, and correct those parts of the code. You should compile the corrected file(s) to make sure that they work; when in doubt about whether your corrected code compiles, we will use `gcc -Wall` to compile the code. Your corrected code should not be subject to any other attacks; for example, don't fix a buffer overflow attack and have the fix be vulnerable to an integer overflow attack. These corrected files should clearly indicate each change with a comment of the form

```
/* corrects [give name] attack */
```

where *[give name]* should be replaced with the name of the attack you have identified. For example, suppose that a problem gives you the following file.

```
#include <stdio.h>
int main() {
 char buf[20];
 gets(buf);
 printf(buf);
 return 0;
}
```

Then a correct solution would be turning in the following file.

```
#include <stdio.h>
int main() {
 char buf[20];
 fgets(buf, sizeof(buf), stdin); /* corrected buffer overflow attack */
 printf("%s", buf);  /* corrected format string attack */
 return 0;
}
```

Note that besides editing code, you can comment it out to delete it.

Problems

For all problems in this homework, there will be a C file specified. The function(s) in this file may have any number of possible vulnerabilities to various attacks. Your task is to find all the vulnerabilities and correct them, producing a functionally equivalent file (for the normal case where the program is not being attacked) that does not suffer from the attacks.

To make the analysis modular, when analyzing a stand-alone function, assume that all arguments to the function have the types specified and that these arguments may be controlled by an attacker. All inputs should also be assumed to be controlled by attackers.
Turn in the corrected file on webcourses, as described in the "what to turn in" section above.

1. (12 points) [SecurelyConstruct] Your task is to find the vulnerabilities in the code in the file `prompt.c` shown in Figure 1 and correct them. The idea of this function is to prompt for user input, but only when the given file (such as standard input) is actually a terminal. Assume that the `pstring` argument may be controlled by an attacker.

```c
/* $Id: prompt.c,v 1.2 2015/10/05 01:36:59 leavens Exp $ */
#include <stdio.h>
#include <string.h>

void prompt(FILE *f, const char *pstring)
     /* requires: f is open */
     /* effect: if f is associated with a terminal,
      *         then write pstring to that terminal (as a prompt).
      */
{
  extern char *ttyname(int fd);
  extern int isatty(int fd);
  char prom[72];

  if (isatty(fileno(f))) {
    /* prompt only if someone to see it */
    FILE *promptf;
    promptf = fopen(ttyname(fileno(f)), "w");
    strcpy(prom, pstring);
    fprintf(promptf, prom);
    fclose(promptf);
  }
}
```

Figure 1: The C file `prompt.c`.

Be sure to turn in corrected code with comments indicating the type of attack you are correcting, and make sure that the code you turn in compiles correctly (as described above).

2. (24 points) [SecurelyConstruct] Your task is to find the vulnerabilities in the code in the file `print_errors.c` shown in Figure 2 on the next page and correct them. This file contains several functions, and attempts to collect logic for printing error messages. Assume that all string arguments may be controlled by an attacker.

   As noted in the code, do not attempt to correct the functions `warning`, `error`, `sys_error`, or `sys_warning`, as these variable-argument functions cannot be easily fixed. They should be considered to be like `printf` in later problems.

   Be sure to turn in corrected code with comments indicating the type of attack you are correcting, and make sure that the code you turn in compiles correctly (as described above).

```c
/* $Id: print_errors.c,v 1.3 2015/10/05 05:06:11 leavens Exp leavens $ */
/* print_errors -- routines to print error messages with name of the program
 * SYNOPSIS: All output is to standard error
 *     program_usage_init(argv[0], "options arguments");
 *     usage();
 *     error("you blew it");
 *     warning("This is your %drd warning", 3);
 *     sys_err("can't open file %s", f);
 *     sys_warning("can't open file %s, continuing", f);
 */
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
static char * Name = NULL;
static char * Usage = "USAGE NOT DEFINED";

void program_usage_init(char *name, char *usage_mesg)
     /* effect: initialize the error printing system */
{
  char * lastslash = strrchr(name, '/');
  Name = (lastslash != NULL) ? lastslash+1 : name;
  Usage = (char *) malloc(strlen(usage_mesg));
  strcpy(Usage, usage_mesg);
}
static void print_name(void)
     /* effect: if Name is defined, print it, a colon, and a space */
{
  if (Name) {
    fprintf(stderr, Name);
    fprintf(stderr, ": ");
  }
}
static void warn_head(void)
     /* effect: if Name is defined, print it, and "Warning: " */
{
  print_name();
  fprintf(stderr, "Warning: ");
}
static void error_head(void)
     /* effect: if Name is defined, print it, and "ERROR: " */
{
  print_name();
  fprintf(stderr, "ERROR: ");
}
void usage()
     /* effect: print usage message on standard error output, and exit(1)
      */
{
  fprintf(stderr, "Usage: ");
  fprintf(stderr, Name);
  fprintf(stderr, " %s\n", Usage);
  exit(1);
}
/* For purposes of this homework, STOP HERE.
```

Figure 2: The important part of the C file `print_errors.c`.

3. (24 points) [SecurelyConstruct] Your task is to find the vulnerabilities in the code in the file `histogram.c` shown in Figure 3 on the following page and correct them. This file contains a program that reads numbers and prints an (ASCII) histogram. It uses the files from the previous two problems for prompting and printing error messages. Assume that all of the program's inputs may be controlled by an attacker.

   Be sure to turn in corrected code for `histogram.c` with comments indicating the type of attack you are correcting, and make sure that the code you turn in compiles correctly (as described above). The `prompt.h` and `print_errors.h` files are included with the zip file for thia homework.

   Hint: You can use your corrected versions of `prompt.c` and `print_errors.c` for testing and compilation purposes.

Points

This homework's total points: 60.

```c
/* $Id: histogram.c,v 1.4 2015/10/05 05:11:12 leavens Exp leavens $ */
/* histogram -- print histogram of raw scores
 * SYNOPSIS: User inputs the scores, one per line on standard input.
 * A histogram is printed on standard output.
 */
#include <stdio.h>
#include <string.h>
#include "prompt.h"
#include "print_errors.h"
#define MAX_SCORE 200
#define HSIZE ((MAX_SCORE)+1)

int readnum(int *temp)
  /* effect: read a number from standard input and place it into temp
     ensures: result is 1 if a number was successfully read into temp,
              and result is 0 otherwise.
  */
{
  char linebuf[80];
  strcpy(linebuf, "\n");  /* initialization */
  gets(linebuf);
  return sscanf(linebuf, "%d", temp);
}
int main(int argc, char *argv[])
{
  int n, i, temp;
  int lowi = HSIZE;   /* index of low score */
  int highi = 0;    /* index of high score */
  int hgram[HSIZE];   /* scores for each student */

  program_usage_init(argv[0], "");
  argc--;
  if (argc >= 1) {
    usage();
  }
  /* initialize hgram */
  for (n = 0; n < HSIZE; n++) {
    hgram[n] = 0;
  }
  /* get scores from standard input */
  prompt(stdin, "scores (1 per line, stop with EOF (Ctrl-D))?\n");
  for(; (readnum(&temp) == 1);) {
    if (0 <= temp) {
      hgram[temp] += 1;
      if (temp < lowi) { lowi = temp; }
      if (temp > highi) { highi = temp; }
    } else {
      fprintf(stderr, "Problem storing into hgram array (at address %p)\n",
              (void *)hgram);
      error("illegal score: %d ", temp);
    }
  }
  /* print histogram, starting with the highest value */
  for (n = highi; n >= lowi; n--) {
    printf("%3d: ", n);
    for (i = 0; i < hgram[n]; i++) {
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```

Figure 3: The C program file `histogram.c`.