Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2012-08-24
Sal	TER3
Tid	14-18
Kurskod	TDDC90
Provkod	TEN1
Kursnamn/benämning	Programvarusäkerhet
Institution	IDA
Antal uppgifter som ingår i tentamen	10
Antal sidor på tentamen (inkl. försättsbladet)	7
Jour/Kursansvarig	David Byers
Telefon under skrivtid	013-282821
Besöker salen ca kl.	14:30, 17:00
Kursadministratör (namn + tfnnr + mailadress)	Madeleine Häger 282360, madha@ida.liu.se
Tillåtna hjälpmedel	Inga
Övrigt (exempel när resultat kan ses på webben, betygsgränser, visning, övriga salar tentan går i m.m.)	

LiTH, Linköpings tekniska högskola
IDA, Institutionen för datavetenskap
Nahid Shahmehri

Written exam

TDDC90 Software Security

2012-08-24

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

David Byers, 013-282821

Instructions

The exam is divided into two parts with a total of ten questions. You should answer all questions in all parts. In order to get the highest grade you will need sufficient points in the second part.

You may answer in Swedish or English.

Grading

Your grade will depend on the total points you score on the exam. The following grading scale is preliminary and might be adjusted during grading.

Grade	3	4	5
Points required	20	27	35

Question 1: Software security (4 points)

State and briefly explain the two principles and/or practices that you think contribute the most to good software security. Rank these principles in order of importance, and motivate your ranking.

You will be graded in part on your priorities and ranking and in part on your explanations and motivation.

Question 2: Vulnerabilities (2 points)

What effect does the choice of programming language have on software security? Can the choice of programming language completely eliminate vulnerabilities? Motivate your answers!

Question 3: Fuzz testing (2 points)

What is white-box fuzz testing? Explain one advantage of white-box fuzz testing over "normal" fuzz testing.

Question 4: Processes (2 points)

Explain the difference between process-specific solutions for developing secure software and security plug-ins.

Question 5: Secure software development (8 points)

Assume that you are developing a web application. What would you do in

a) the design phase or its equivalent (also name design principles or patterns you would follow)

b) the implementation phase or its equivalent

to ensure that your software is free of (exploitable) buffer overflow vulnerabilities? Motivate your answer. You will be graded both on the selection of activities (appropriateness, completeness) and ability to explain why these activities/measures were chosen.

Question 6: Attack Trees (4 points)

An advertising company, Spam'R'Us, decides to steal a large number of e-mail addresses from a popular social network. Draw an attack tree for this attack scenario! Also, briefly explain what attack trees are used for.

Question 7: Static analysis (2 points)

Name and briefly explain one advantage and one drawback of static analysis over normal software security testing.

Question 8: Secure design patterns (4 points)

Explain a secure design pattern of your choice, including its purpose, the pattern itself, and what security benefits it provides.

Question 9: Secure software engineering (4 points)

Name and briefly explain two security activities in SDL that take place in the design phase.

Question 10: Vulnerabilities (12 points)

The program, *readlog*, on the next page is used to allow regular users access to read the last 40 lines of log files they would normally not have access to. The program is run with a single argument, the name of a file in the /var/log directory that the user wants to view.

Access is given to files only if the group of the file is *logger* and the group has read permissions to the file. Since *readlog* is installed setuid root, the user executing the program does not need to be a member of the *logger* group.

Additionally, the *readlog* program must log every access to a log file in its own log file, /var/log/readlog.log; readlog.log must provide an accurate record of which users have been granted access to which log file: under no circumstances may a user use *readlog* to access a log file without this being recorded (however, if *readlog* does not give access, then nothing needs to be recorded).

The program works well and compiles without warnings in gcc –W –Wall (this turns on all relevant warnings). However, the code contains several vulnerabilities.

For at least two security vulnerabilities **of different kinds**:

- Indicate the code that contains the vulnerability.
- Explain what input might trigger the vulnerability and very roughly how the vulnerability could be exploited.
- Propose corrections to the code that would eliminate the vulnerability. If you are unable to write actual code, then write well-explained pseudocode. You need to show that you understand how the problem could be fixed.
- Name and explain any mitigation techniques in the compiler, libraries or operating system that could prevent at least one of the vulnerabilities from being exploited.

Finally, the code violates one fundamental security design principle that is not reflected in the vulnerabilities in the code. Name and explain that principle.

Code for question 10

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>

#define LOG_LOG "/var/log/readlog.log"
#define LOG_PREFIX "/var/log"
#define LOG_GROUP "logger"

int main(int argc, char **argv) {
  char path[MAXPATHLEN], cmd[MAXPATHLEN + 10];
  struct stat statbuf;
  struct passwd *pw;
  struct group *gr;
  FILE *fp;
  time_t now = time(NULL);

  if (!argv[1]) exit(1);        /* No argument given to program */

  // Get information about the "logger" group
  if ((gr = getgrnam(LOG_GROUP)) == NULL)
    exit(1);                    /* Group didn't exist */

  // Build the path to the log file to show
  sprintf(path, "%s/%s", LOG_PREFIX, argv[1]);

  // Check if access to file is permitted
  if ((stat(path, &statbuf) != 0)     || /* No access to file */
      (statbuf.st_gid != gr->gr_gid) || /* Wrong group for file */
      !(statbuf.st_mode & S_IRGRP))     /* Wrong permissions */
    exit(1);

  // Access to the file is permitted
  if ((pw = getpwuid(getuid())) != NULL)
    exit(1);                    /* Calling user doesn't exist */

  // Log access to the readlog log
  if ((fp = fopen(LOG_LOG, "ab")) == NULL)
    exit(1);                    /* Logger log didn't exist */
  if (fprintf(fp, "%s %s read %s\n", ctime(&now),
                        pw->pw_name, argv[1]) <= 0)
    exit(1);                    /* Failed to write to logger log */
  fclose(fp);

  // Show the file using the "cat" command
  sprintf(cmd, "tail -40 %s", path);
  system(cmd);

  exit(0);
}
```

Notes on the code for those not very familiar with C

In several places, the code uses a pattern similar to this:

```
if ((var = func(...)) == NULL)
```

This is shorthand for:

```
var = func(...)
if (var == NULL)
```

This is possible since assignments are simply expressions in C, and return the value that was assigned. Naturally other comparisons than *equals NULL* are permitted (and used).

The code uses the following standard C library and Unix functions:

stat gets information about a file such as its owner, permissions and so forth. In the returned structure, the field st_gid holds the group ID of the file and st_mode holds the permissions. The bit S_IRGRP is set to 1 if the file is readable by the group (so *statbuf.st_mode & S_IRGRP* yields 1 if the group read permission bit is set).

getpwuid(*uid*) gets information about the user with user ID *uid*. In this program, we only use the user's name. **getgrnam(*name*)** gets information about a group with name *name*. In this program, we then use the group number, stored in the gr_gid field.

fopen(*path, mode*) opens the file *path*, returning a file pointer. The *mode* ab means open the file for appending in binary mode. This is the correct mode for this program.

fprintf(*fp, fmt, ...*) outputs a formatted string to the open file pointed to by *fp*. If the file failed to open, the program will terminate. On failure, this function returns -1. Otherwise it returns the number of characters printed. **sprintf(*buf, fmt, ...*)** outputs a formatted string into the buffer *buf*. This function can be assumed to never fail to write the requested data to the buffer.

Both these functions output the format string (fmt), with the character sequence %s replaced by the remaining arguments, one at a time. For example, *fprintf(fp, "%s and %s", "one", "two")* will output the string "one and two".

system(*cmd*) executes the command *cmd* in a shell (in practice, it starts /bin/sh and executes the command in the resulting shell).

fclose(*fp*) closes the open file pointer *fp*. All open file pointers are also automatically closed on program exit.

exit(*status*) closes all files and terminates the program with exit status *status*.

MAXPATHLEN is the longest a path name (i.e. a filename with complete specification of directories) can be in Unix. File-related functions will not accept longer paths.

Strings in C are terminated by ASCII NUL (0), i.e. *null terminated*. Unless otherwise specified, all functions will return or produce correctly terminated strings.