Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-01-15
Sal	KÅRA
Tid	8-12
Kurskod	TDDC90
Provkod	TEN1
Kursnamn/benämning	Programvarusäkerhet
Institution	IDA
Antal uppgifter som ingår i tentamen	9
Antal sidor på tentamen (inkl. försättsbladet)	7
Jour/Kursansvarig	David Byers
Telefon under skrivtid	013-282821
Besöker salen ca kl.	9:00, 11:15
Kursadministratör (namn + tfnnr + mailadress)	Madeleine Häger 282360, madha@ida.liu.se
Tillåtna hjälpmedel	Inga

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

Written exam

TDDC90 Software Security

2013-01-15

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

David Byers, 013-282821

Instructions and grading

You may answer in Swedish or English.

Your grade will depend on the total points you score on the exam. The maximum number of points is 46. The following grading scale is preliminary and might be adjusted during grading.

Grade	3	4	5
Points required	22	29	36

Question 1: Processes (4 points)

To include security in software development processes you can use (1) a security plug-in, (2) a process with security included from the beginning or (3) ad-hoc application of best practices. Which one of the three alternatives would you use for:

a) A small-scale hobby project you plan to start with a friend?

b) Large-scale development of the next version of an existing software product with million lines of code?

You must motivate your answers to get any points on this question.

Question 2: Integer overflows (4 points)

In this question you may choose between command injection and SQL injection (you must choose the same in both part a and in part b). You do not need to answer for both command *and* SQL injection.

a) Explain what a command or SQL injection vulnerability is.

b) Give a concrete example of code containing a command or SQL injection vulnerability.

Question 3: Fuzz testing (8 points)

Fuzz testing has several important challenges. Name two important challenges of fuzz testing. For each challenge, explain what the challenge is, and how it can be addressed.

For example:

> *A challenge of fuzz testing is finding suitable input to modify. One way of fuzzing is to take a valid input and alter it. This means you need a valid input to begin with, and the challenge is finding suitable valid inputs. This can be addressed by fuzzing without valid inputs, or finding e.g. standardized test vectors for the software being fuzzed.*

Obviously, you can't use that challenge as one of your answers.

Question 4: Risk (6 points)

a) Define what a risk is.

b) Briefly describe how threats and vulnerabilities relate to risks.

c) What is the difference between risk management and risk analysis?

d) Do risks always need to be mitigated? Why/why not? Motivate!

Question 5: Security requirements (2 points)

Bad Software Inc. uses the following security requirements. Name at least one problem with each of them:

a) Use strcpy instead of strncpy to protect against buffer overflows.

b) The code must not have any vulnerabilities.

Question 6: Static analysis (6 points)

a) Give a brief definition: What is static analysis?

b) Give two examples of types of security vulnerabilities easily found with static analysis!

c) Give two examples of types of security vulnerabilities hard to find with static analysis!

Question 7: Security modeling (4 points)

Describe a security modeling method of your choice! What do the models include in terms of objects, actions and actors? What is the modeling method used for? Feel free to draw an example of a model belonging to the method!

Question 8: Secure software engineering (2 points)

Give examples of two software security touch points: one to use early in the software life cycle and one to use in the end!

Question 10: Vulnerabilities and detection (10 points)

The purpose of the program shown on the next page of this exam is to allow a non-privileged user to append data to specific files the user would normally not be permitted to alter (see the program notes, below). Each output line is suffixed with the user's user name.

The following security requirements exist:

- The program must not contain any buffer overflows, format string errors, race conditions, integer overflows or other common vulnerabilities.
- If the program exists with exit status 0 (success), it must have written every byte of input, apart from newlines (which are optional), to the output file.
- The program must not allow the user to write to any other files than those with the appropriate permissions (see *program notes* below for details).
- The program must not crash on any input.

The program has three known vulnerabilities that an attacker *Contines on next page*

a) Identify and explain at least two vulnerabilities in the code. For each vulnerability, indicate the line(s) of code involved, explain how they result in a vulnerability, explain the inputs or actions that would trigger the vulnerability (you do not need to provide a complete exploit), and propose corrections to the code that would eliminate the vulnerability.

b) The program violates one secure design principle. Which one? Explain the principle, and propose modifications to the code so that the principle is no longer violated.

Program notes (you will need these to answer the questions)

The program is installed setuid root (i.e. the effective user ID on execution is 0, which allows the program to access any file). It has to be setuid root in order to do its job, but checks that it only alters files it is allowed to alter.

A user may append to a file provided that the following conditions are met: it is a regular file, not executable by anyone, and has both the setgid and sticky bits set. You shall assume that the program checks these conditions correctly (i.e. the *is_appendable* function is correctly implemented).

The program compiles without any warnings.

There are some extra notes on the various functions used in the code on the last page of this exam.

Code for question 10

```
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <pwd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```
/*
 * Check whether PATH is a file we are allowed to append data to. A
 * file is appendable if (and only if) it is writeable by the owner,
 * has the setgid and sticky bits set, and is not executable.
 */

int is_appendable(char *path) {
    struct stat statbuf;

    if (stat(path, &statbuf))          /* Get metadata for the file */
        return 0;                      /* If that fails, return 0 */
    if (S_ISREG(statbuf.st_mode) &&    /* Is it a regular file? */
        (statbuf.st_mode & S_ISGID) && /* Is the setgid bit set? */
        (statbuf.st_mode & S_ISVTX) && /* Is the setgid bit set? */
       !(statbuf.st_mode & S_IXUSR) && /* The file may not be user ... */
       !(statbuf.st_mode & S_IXGRP) && /* ... group ... */
       !(statbuf.st_mode & S_IXOTH) && /* ... or other executable */
        (statbuf.st_mode & S_IWUSR))   /* Is it user-writeable? */
        return 1;
    return 0;
}

#define BUFSZ 2048              /* Ensure consistent buffer size */

int main(int argc, char **argv) {
    char outb[BUFSZ], inb[BUFSZ];
    struct passwd *pw;
    int ulen;
    FILE *fp;

    if (!argv[1]) exit(1);                  /* Exit if no filename given */

    /* Get some information about the user so we can log that to the
       output file together with the user's data. */

    pw = getpwuid(getuid());                /* Get user information */
    if (!pw) exit(1);                       /* Exit if getpwuid failed */
    ulen = strlen(pw->pw_name);             /* Get username length */

    /* Check that the target file is appendable. */

    if (is_appendable(argv[1])) {

        /* Now that we know that it is safe to write to the file, we
           open it for writing (in binary mode), and seek to the end
           of the file, to ensure that we are appending, not writing
           somewhere in the middle. */

        fp = fopen(argv[1], "ab");           /* Yes, so open it */
        if (!fp) exit(1);                    /* If open fails, then exit */

        /* Read one line at a time until the end of file. We read at
           most BUFSZ-ulen characters in order to be sure that the
           username (ulen chars) and the read line will fit into the
           target buffer. */

        while (fgets(inb, BUFSZ - ulen, stdin)) {

            /* If there is a trailing newline, then remove it */

            if (inb[strlen(inb)-1] == '\n')
                inb[strlen(inb)-1] = '\0';

            /* Build the output string. We checked that there is
               enough room, but will still use strncpy/strncat. First
               we copy the user name, then a separator, then the
               user's input. In each step we copy at most the number
               of characters remaining in the buffer. */
```

Contines on next page

```
            strncpy(outb, inb, BUFSZ - ulen - 4)
            strncat(outb, " [", BUFSZ-strlen(outb));
            strncat(outb, pw->pw_name, BUFSZ-strlen(outb));
            strncat(outb, "]\n", BUFSZ-strlen(outb));

            /* Write the line to the output file; if writing fails,
               exit with a non-zero exit status. */

            if (fputs(outb, fp) == EOF)
                exit(1);
        }
        fclose(fp);                          /* Close the output file */
    }
    exit(0);                          /* Exit the program with success */
}
```

Contines on next page

Notes on the code for those not very familiar with C

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. Note that you shall assume that is_appendable is correctly implemented.

getpwuid(*uid*) gets information about the user with user ID *uid*. In this program, we only use the user's name.

getuid() returns the *real* user ID of the process, i.e. the user that invoked the program. So even if the program is setuid root, this will return the ID of the user who started it.

strlen(*str*) returns the number of characters in the string *str*. Thus, *str[strlen(str)-1]* is the last character in the string *str*.

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.

fgets(*buf*, *bufsz*, *fp*) reads one line of input from the specified file. The *buf* parameter is the buffer to read input to. The *bufsz* parameter is the maximum number of bytes to write to the buffer. The *fp* argument is the file pointer to read from. If *fgets* successfully reads an entire line, it will include the terminating linefeed character in the string. Under no circumstances will *fgets* write more than *bufsz* bytes to the target buffer.

strncpy(*dst*, *src*, *n*) copies at most *n* characters from *src* to *dst*. If there is no null (zero) byte among the first *n* characters of *src*, the resulting string will not be null terminated.

strncat(*dst*, *src*, *n*) appends at most *n* characters from *src* to *dst*, overwriting the null byte at the end of *dst*, then adds a terminating null byte after the appended characters.

fputs(*buf*, *fp*) writes the contents of *buf* to the file pointer *fp*.

fclose(*fp*) closes the open file pointer *fp*.

exit(*status*) terminates the program with exit status *status*.