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

Written exam

TDDC90 Software Security

2011-04-26

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Shanai Ardi / 282608

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 (2 points)

State and briefly explain the two principles and/or practices that you think contribute the most to good software security. You will be graded in part on your priorities and in part on your explanations.

Question 2: Vulnerabilities (2 points)

Briefly explain one compiler or linker based method for preventing exploitation of stack-based buffer overflows.

Question 3: Fuzz testing (4 points)

a. Explain what fuzz testing is and how it works.
b. What kind of security problems is fuzz testing useful for?
c. Give an example of a security problem that fuzz testing will probably not detect, and suggest an activity (e.g. testing or analysis) that would be appropriate for detecting that kind of problem. Motivate your answer.

Question 4: Common criteria (2 points)

Explain what a security target is and how it is used in the Common Criteria.

Question 5: Vulnerabilities (2 points)

Explain what an integer overflow is. Give a concrete example of how an integer overflow can lead to a security vulnerability.

Question 6: Security requirements (4 points)

Briefly explain the Security Quality Requirements Engineering (SQUARE) methodology.

Question 7: Static analysis (6 points)

Static analysis can be applied to both source code and executables (machine code, byte code etc.).

a. Briefly discuss the advantages and disadvantages of both approaches.

b. Give a concrete example of a situation where analysis of source code would be better than analysis of the executable.

c. Give a concrete example of a situation where analysis of the executable would be better than analysis of the source code.

Static analysis can be applied to executables as well as source code. Explain why it might be better, in some situations, to analyze the executable. Include a concrete example to illustrate your explanation. Also explain why analysis of source code may be preferred (or even necessary) in other situations.

Question 8: Secure design patterns (2 points)

What is a secure design pattern? Why would a developer use a secure design pattern rather than create a custom design?

Question 9: Secure software engineering (6 points)

Explain what misuse cases are. Draw a misuse case diagram and create at least one detailed misuse case for an on-line course database where users can register for courses and check their results, and teachers can register results (much like IDA's webreg system).

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 could exploit.

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. 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];
```

Contines on next page

```
    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. */

            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, which is known to be at most 64 characters, printable ASCII characters only.

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*, not counting the terminating ASCII NUL character. 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*.

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.