Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2009-08-24
Sal	T1
Tid	14-18
Kurskod	TDDC90
Provkod	TEN1
Kursnamn/benämning	Software Security
Institution	IDA
Antal uppgifter som ingår i tentamen	10
Antal sidor på tentamen (inkl. försättsbladet)	4
Jour/Kursansvarig	David Byers/Nahid Shahmehri
Telefon under skrivtid	013-282821 / 0708-282821
Besöker salen ca kl.	15:00, 16:30
Kursadministratör (namn + tfnnr + mailadress)	Madeleine Häger 282360, madha@ida.liu.se
Tillåtna hjälpmedel	Tryckt ordlista
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

2009-08-24

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

David Byers, 013-282821 / 0708-282821

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	18	24	30

Important
In order to get the highest grade you must have scored at least six points in part 2.

Part one

Question 1: Privilege Separation (2 points)

Explain what privilege separation is, and how it helps improve security in software.

Question 2: The Common Criteria (2 points)

What is the difference between Security Target and Protection Profile in Common Criteria?

Question 3: Static Analysis (2 points)

Explain at least one good point ("pro") and one bad point ("con") of static analysis.

Question 4: Developing Secure Software (2 points)

Is it possible for a program to have serious security flaws despite the implementation being absolutely flawless? Motivate your answer.

Question 5: Fuzz Testing (4 points)

The following C code calculates the line equation (in the form $y = ax + b$) for the line that intersects two points p1 and p2. For some inputs, the program will crash.

```
struct line { int a, b; };
struct point { int x, y; };

struct line *create_line(struct point p1, struct point p2) {
    struct line *l = malloc(sizeof(struct line));
    l.a = (p2.y - p1.y) / (p2.x - p1.x);
    l.b = p1.y - l.a * p1.x;
    return l;
}
```

Is fuzz testing using random inputs for p1 and p2 likely to detect the problem? Motivate your answer. If your conclusion is that fuzz testing is unlikely to detect the problem, then discuss how this *kind* of problem could be overcome in fuzz testing.

Question 6: Security Requirements (4 points)

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

Question 7: Race Conditions (4 points)

Explain what a race condition is, and how it relates to software security. Give two different code examples showing (potential) race conditions, and explain how each works, and how each could be exploited.

Question 8: Common Criteria (4 points)

What are Evaluation Assurance Levels in Common Criteria and how are they used? If you use any further terminology from the Common Criteria, briefly explain each term.

Part two

In order to score well on these questions you will need to show that you understand not only the technical issue or concept at hand, but also its context and its interactions with its context (e.g. processes, methods, techniques, technology, people, risks, threats, and so on). We *think* that good answers to these questions will require at least one or two handwritten pages (more or less may be required depending on how you write).

Question 9: RMF (6 points)

The Secure Development Lifecycle (SDL) is an increasingly popular approach to secure software development. Explain how it works. What SSE-CMM capability level do you think this process belongs to? Motivate your answer.

Question 10: Vulnerabilities (6 points)

The function shown on the last page of this exam (read_ppm) parses a portable pixmap image file. A PPM file consists of a "magic number", followed by the image dimensions, color depth, and finally the image data. Each pixel of image data is either three bytes or six bytes long, depending on the color depth.

The read_ppm function below works quite well for normal PPM files, but contains at least two vulnerabilities that can be exploited using files with carefully chosen contents.

For each vulnerability:

- Indicate the code that contains the vulnerability.
- Explain the input that could trigger the vulnerability (you do not need to explain how to exploit it).
- Propose corrections to the code that would eliminate the vulnerability.
- Name and explain any mitigation techniques in the compiler, libraries or operating system that could prevent the vulnerabilities from being exploited.

Code for question 10

```
struct image *read_ppm(FILE *fp)
{
    int version;
    int rows, cols, maxval;
    int pixBytes=0, rowBytes=0, rasterBytes;
    uint8_t *p;
    struct image *img;

    /* Read the magic number from the file */
    if ((fscanf(fp, " P%d ", &version) < 1) || (version != 6)) {
        return NULL;
    }

    /* Read the image dimensions and color depth from the file */
    if (fscanf(fp, " %d %d %d ", &cols, &rows, &maxval) < 3) {
        return NULL;
    }

    /* Calculate some sizes */
    pixBytes = (maxval > 255) ? 6 : 3; // Bytes per pixel
    rowBytes = pixBytes * cols;        // Bytes per row
    rasterBytes = rowBytes * rows;     // Bytes for the whole image

    /* Allocate the image structure and initialize its fields */
    img = malloc(sizeof(*img));
    if (img == NULL) return NULL;
    img->rows = rows;
    img->cols = cols;
    img->depth = (maxval > 255) ? 2 : 1;
    img->raster = (void*)malloc(rasterBytes);

    /* Get a pointer to the first pixel in the raster data. */
    /* It is to this pointer that all image data will be written. */
    p = img->raster;

    /* Iterate over the rows in the file */
    while (rows--) {
        /* Iterate over the columns in the file */
        cols = img->cols;
        while (cols--) {
            /* Try to read a single pixel from the file */
            if (fread(p, pixBytes, 1, fp) < 1) {
                /* If the read fails, free memory and return */
                free(img->raster);
                free(img);
                return NULL;
            }

            /* Advance the pointer to the next location to which we
               should read a single pixel. */
            p += pixBytes;
        }
    }

    /* Return the image */
    return img;
}
```

Notes on the code for those not very familiar with C

fscanf reads from a file to a program variable. The second argument specifies the input format. For example, `fscanf(fp, " P%d ", &version)` reads zero or more whitespace characters followed by an uppercase "P", followed by an integer. The value of the integer is stored in the variable named *version*. The fscanf function returns the number of items it read successfully.

malloc allocates memory on the heap. The parameter to malloc specifies how much memory can be allocated. Memory allocated with malloc is returned to the heap using the **free** function. When malloc fails to allocate sufficient memory, it returns NULL.

The data type **image** has four fields, rows, cols, depth and raster. When initialized, rows contains the number of rows in the image, cols the number of columns, and depth is either 1 or 2, indicating how many bytes are used to represent a single color value. The raster field contains a pointer to a memory area on the heap that holds all the pixel values. Each pixel value is a sequence of three color values.

The **fread** function reads raw bytes from a file. In this function it is used to read the image data, one pixel (i.e. 3 or 6 bytes) at a time. It returns the number of items read (in this function, the number of pixels). Specifically, the function call `fread(p, pixBytes, 1, fp)` will read *one* item that is *pixBytes* long from the file pointed to by *fp*, and store that item in the memory location pointed to by the variable *p*.

The **sizeof** operator returns the size of something. For example, `sizeof(int)` will return the number of byte required to store an integer, and assuming *img* is a pointer to a `struct image`, `sizeof(*img)` will return the number of bytes required to store an image structure.