Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2012-12-22
Sal	U1, U3, U4
Tid	8-12
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	Anna Vapen / Nahid Shahmehri
Telefon under skrivtid	073-8491275
Besöker salen ca kl.	9, 11
Kursadministratör (namn + tfnnr + mailadress)	Madeleine Häger 282360, madha@ida.liu.se
Tillåtna hjälpmedel	Ordbok (ej elektronisk)

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

Written exam

TDDC90 Software Security

2012-12-22

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Anna Vapen, 073-8491275

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 44. The following grading scale is preliminary and might be adjusted during grading.

Grade	**3**	**4**	**5**
Points required	20	27	35

Question 1: Processes (2 points)

What is the difference between a security plug-in and a process specific solution, in the field of secure software development processes?

Question 2: Vulnerabilities (2 points)

Can there be security problems with a program that is completely free of vulnerabilities? Motivate your answer!

Question 3: Fuzz testing (4 points)

Write one or two function(s) in the language(s) of your choice (or pseudo-code) containing one vulnerability that typical fuzz-testing would easily find, and one that typical fuzz-testing would be unlikely to find. Explain why these vulnerabilities would be easy/hard to find using fuzz testing.

Question 4: Risk (6 points)

a) Define the term *risk*!

b) Is it necessary to mitigate all risks found during a risk analysis? Why or why not? Motivate your answer!

c) Briefly explain what RMF is and what it is used for.

Question 5: Security requirements (4 points)

As a developer at SecureSoft Inc, you are given the following security requirements, regarding the software you are developing:

No.	Name	Description
1	Downtime	The servers shall never be down.
2	Vulnerability mitigation	Use SQUARE to prevent vulnerabilities in the software.
3	String handling	Use strcpy instead of strncpy to prevent buffer overflows.

The requirements are given to you during the maintenance phase in the software lifecycle. You are not entirely happy either with the timing or the set of requirements.

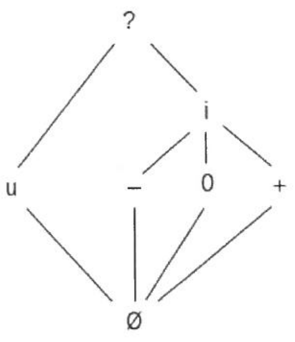
a) When in the software lifecycle should security requirements be introduced?

b) Explain and motivate one problem with each requirement.

Question 6: Static analysis (4 points)

Assume that a static analyzer includes checking for uninitialized data. It should find the cases where a number is the value of such an uninitialized variable, or was computed using such a value. Assume that the analyzer employs following abstract values:

Abstract value	**Denotes**
(?)	any
(u)	uninitialized
(i)	the set of integers
(+)	the set of positive integers
(0)	the set {0}
(-)	the set of negative integers
(Ø)	the empty set



Abstract value (u) corresponds to the set of integers, but also informs that the number is the value of an uninitialized variable, or was computed using such value. Also, (i), (+), (0), (-) mean that the corresponding value was computed in a "legal way", and does not depend on uninitialized data. Note that 0 is neither positive, nor negative. For instance in this code fragment:

```
int a, b, c;
a = 0;
b = a+c;
```

after executing the two assignments, the variable *a* is initialized, *b* is uninitialized, and *c* is uninitialized, as a result of addition involving uninitialized *c*. (The value of *b* or *c* corresponds to (u) and to (?), and that of *a* to (0), (i), (?)).

Consider a code fragment, which is executed with initially uninitialized variable *y* (abstract value (u)), and the remaining variables initialized (abstract value (i)).

```
while (x >= 0) { ... /* y does not occur here */ }
if (y>3) { z=3; }
if (x<0) { y=0; }
```

What can a sound static analyzer find out about the value of *x*, *y* and *z* at the end of the code fragment?

Question 7: Design patterns (2 points)

Briefly explain the design pattern "privilege separation" and what it accomplishes from a security perspective.

Question 8: Security modeling (6 points)

Explain the following three security modeling techniques and describe what they are used for, when in the software lifecycle they should be used and what they contain (e.g. vulnerabilities, risks, requirements, attacks, threats or something else). Motivate your answers!

a) Attack tree

b) Misuse case

c) VCG (Vulnerability Cause Graph)

Question 9: Secure software engineering (2 points)

Name and briefly explain two security touchpoints (i.e. activities or artifacts used to improve security during the software lifecycle). Explain where in the software lifecycle the two touchpoints belong.

Question 10: Vulnerabilities (12 points)

The function shown on the next 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

```c
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.