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

Written exam

TDDC90 Software Security

2013-03-25

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 (2 points)

Give an example of a software development process and explain in a few sentences how security is included in the process (i.e. as part of the process, as a process of its own or in any other way). If security is not included in the process, explain how you would choose to include it and why you would do it that way.

Question 2: Integer overflows (6 points)

Integer overflows are a special case of arithmetic errors.

a) Explain what an integer overflow is, and how it works. Explain how integer overflows can result in security vulnerabilities.

b) Give a concrete example of code containing an integer overflow that can be exploited from a security point of view. Give at least one (set of) inputs that will trigger the vulnerability.

Question 3: Fuzz testing (6 points)

Imagine that you are writing a fuzzer that can fuzz integers and strings, and your goal is to find buffer errors.

a) Describe how you would chooses/generate the strings to test.
b) Describe how you could choose/generate the integers to test.

Be precise. From your answers it should be possible to derive which specific strings and values that your fuzzer would generate. Motivate your answers!

Question 4: Risk (6 points)

You work at a small company developing an online game which is accessible to users via a website. Users log in to the website with a username and password. It is possible for users to buy additional equipment for their game characters. To do so, a user needs to provide their credit card number.

a) Name two risks connected to the online game. If you could only mitigate one of the risks, which one would you choose and why? *State any assumptions you make!*

b) Name and explain two methods, frameworks or similar for risk analysis and/or risk management. For each method, explain if it is suitable to use it in this specific context. Why or why not? *Again, state your assumptions!*

Question 5: Security requirements (4 points)

Explain what a security requirement is and where in the software lifecycle it is suitable to handle security requirements. Give an example of a methodology for security requirements engineering and briefly explain it.

Question 6: Static analysis (6 points)

a) Explain what it means that a static analysis is sound.

b) What can you say about soundness of industrial static analysis tools?

c) Describe how static analysis differs from dynamic analysis and testing.

Question 7: Security modeling (6 points)

a) What is an attack tree and what is it used for? Sometimes, cost is added to the branches of the attack tree. Do you think this is a good idea or not? Why or why not?

b) Explain briefly what a misuse case is. Is it possible to create a misuse case out of the information provided in an attack tree? Explain why or why not!

c) Explain another way (than the one described in b) to create a misuse case. Also explain why or why not this other way is better than starting with the attack tree.

Question 8: Secure software engineering (2 points)

What is an artifact in the context of secure software engineering? Name two artifacts and place them in the software development lifecycle.

Question 9: Vulnerabilities (8 points)

The function shown on the next page of this exam is a simple request handler for a web application server. The request handler is called by the application server for specific requests. You don't need to be concerned about how this works.

This particular request handler is for file uploads. The request contains two important parts: a *path* and *data*. Both can be accessed via a request object, which the request handler gets from the application server.

The path indicates where to store the uploaded file. To prevent malicious users from overwriting arbitrary files on the computer, the request handler prepends a *document root* to the requested path. For example, if the request specified path `/etc/passwd`, and the document root is `/uploads`, then the request handler will store the uploaded data in the file `/uploads/etc/passwd`. It is important that no files are ever stored outside the document root.

The data is the data to upload. It is assumed to be text encoded using ISO-8859-1, which means that there is one byte per character, of which all eight bits are significant. The request handler reads all the data into memory, converting one character at a time to UCS-4, which uses exactly 32 bits per character. The converted data is then written to the output file.

The request handler requires the session to be authenticated.

There are at least two vulnerabilities in the code.

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.

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

Continues on next page

Code for question 9

```
int request_handler(struct http_request *sess) {
    char anonymous;
    char path[MAXPATHLEN];
    int size;
    char c, rootd;
    FILE *in, *out;
    uint32_t *buf, *tmp;

    anonymous = is_anonymous(sess);

    /* Check if the request is valid */
    if (sess->request == NULL)
        return INVALID_REQUEST;

    /* Place the document root into path */
    strcpy(path, document_root);

    /* Set rootd to 1 if path is "/" */
    rootd = (path[0] == '/' && path[1] == '\0');

    /* Check that root, request, null and possible extra "/" fits in path */
    if (strlen(path) + strlen(sess->request) + rootd + 1 > MAXPATHLEN)
        return INVALID_REQUEST;

    /* Now we know there is enough space in path. Perform the append */
    if (rootd == 0)
        strcat(path, "/");              /* Add a / if path is not "/" */
    strcat(path, sess->request);        /* Append the request path */

    /* Read, encode, and copy the input if the user is authorized */
    if (!anonymous) {
        size = atoi(http_get_header(sess, "content-length"));
        buf = malloc(size * 4);         /* Space for UCS-4 encoding */
        tmp = buf;                      /* Save a copy of the pointer */

        in = http_get_input_stream(sess);
        while (size--) {                /* Read at most size bytes */
            c = fgetc(in);              /* Get one character */
            if (c == -1)                /* End of file */
                break;                  /* Terminate reading */
            *tmp = latin1_to_ucs4(c); /* Convert character */
            tmp += 1;                   /* Advance to next position */
        }
        fclose(in);                     /* Close the input */

        size = atoi(http_get_header(sess, "content-length"));
        out = fopen(path, "w");         /* Open the output file */
        fwrite(buf, 4, size, out);      /* Write the entire buffer contents */
        fclose(out);                    /* Close the output file */
        free(buf);                      /* Free allocated memory */
        return OK;
    }
    else
        return UNAUTHORIZED;
}
```

Continues on next page

Notes on the code for those not very familiar with C

The code above uses some API functions and variables from the application server:

is_anonymous returns 1 if the request is anonymous (i.e. not authenticated).

http_get_header returns the content of the specified HTTP header.

http_get_input_stream returns a file pointer from which the handler can read the request data. The file pointer returned by this function should be closed using *fclose*.

latin1_to_ucs4 converts a single character from ISO-8859-1 encoding to UCS-4 encoding (i.e. from one to four bytes).

document_root is a string guaranteed to be a valid path on the filesystem, and guaranteed to be no more than MAXPATHLEN characters long.

INVALID_REQUEST, UNAUTHORIZED, and OK are constants that this function may return.

struct http_request represents an HTTP request. The *request* field contains the path the client has requested.

The code also uses the following standard C library functions:

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.

free frees allocated memory. It must never be called twice on the same pointer.

The **fgetc** function reads a single character from a file pointer. It returns an integer representing the character, or -1 if there are no more characters to read.

strcpy copies data to a destination from a source. It operates on null-terminated strings (i.e. standard C strings). For example, to copy a string from a to b, call *strcpy(b,a)*. Both *a* and *b* must be pointers to strings or be character arrays. If *b* contains the string "test", then the function will copy five bytes: the four characters and the null terminator.

strcat concatenates two strings. Like strcpy it operates on standard C strings. For example, to place the contents of *a* at the end of *b*, call *strcat(b,a)*. The resulting string will also be null terminated.

strlen calculates the number of characters in a string. It does not count the null terminator.

atoi converts a string to an integer. If the string does not represent a valid integer, then its behavior is undefined (it will probably return 0).

fwrite writes output to a file pointer. The call *fwrite(buf,size,nitems,fp)* writes *nitems* items of size *size* from the memory that *buf* points to, to the file pointer *fp*.

fclose closes an open file pointer.

uint32_t is an integer datatype that occupies exactly 32 bits. **char** is an integer datatype that represents an ASCII character; it occupies one byte (eight bits). **FILE*** is a *file pointer*, from which functions such as *fgetc* can read input.

MAXPATHLEN is the maximum length of a valid path name.

C handles arithmetic on pointers differently from arithmetic on integers. If *p* is a pointer to a datatype that occupies *n* bytes, then the statement *p += 1* will advance *p* to the next element – i.e. increment it by *n*. In this example, pointer arithmetic is used to advance the *tmp* pointer one element at a time from the start of *buf* up to the last element.