Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-08-26
Sal	TER3
Tid	14-18
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.	15:30
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

2013-08-26

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)

There are several development processes aimed at developing secure software. Name one such process and explain in detail how it works and if it is used on its own or together with any other development process (name which one!).

Question 2: Buffer overflows (6 points)

Explain the following compiler-based mechanisms to protect against buffer overflows:

a) Stack canaries.

b) Reordering of local variables.

Your explanation must cover how the mechanism works, how it protects against buffer overflows, and any potential limitations of the mechanism. Illustrate how each mechanism alters the stack, and how these changes prevent exploiting buffer overflows.

Question 3: Fuzz testing (6 points)

Consider an application that handles a communication protocol whose messages have the following format:

Message type:	2 bytes
TLV count:	2 bytes (number of TLV records to follow)
Data:	Sequence of TLV records
Checksum:	CRC32 checksum of entire message

Each TLV record has the following format:

Type:	1 byte
Length:	2 bytes (length of data to follow)
Data:	*Variable length*

There are two message types: *send* (1) and *receive* (2). If the message type is *send*, then there must be two TLVs: *filename* (1) and *data* (16). The data in the *filename* TLV must consist of alphanumeric characters only. The checksum must be correct, or the message will be

Fuzz testing this application effectively poses several challenges beyond those of fuzzing just one integer or string. Discuss what those challenges are and how they could be solved.

Question 4: Risk (4 points)

When working with risks in a software security context, is it important to always mitigate all risks or are there risks which should not be mitigated? What is a good way of choosing which risks to mitigate (if not all of them)? Motivate your answers!

Question 5: Security requirements (2 points)

In a requirement specification you find the following requirement "There shall not be any buffer overflow vulnerabilities in the code". Explain what is wrong with this requirement and why!

Question 6: Static analysis (6 points)

a) Explain briefly what static analysis is and what it is used for.

b) Give pros and cons (at least two each) for static analysis compared to testing.

c) Give two examples of security vulnerabilities that can be discovered with static analysis.

Question 7: Security modeling (8 points)

Assume there is a website on which students can search for courses and see what other students think about a specific course. Without logging in to the website a student can read course reviews written by other students. If logged in to the website a student can also post their own reviews. A moderator can, when logged in, remove any review on the website.

The infamous Dr Evil realizes that one of his courses has gotten bad reviews by his students on the website described above. He decides to attack the website and remove all negative reviews.

a) Draw a detailed misuse case describing the website, the normal usage of the site (by students and the moderator), and Dr Evil's attack against it! Be clear about how the attack works in detail. Write down all assumptions you make. (6p)

b) Draw an attack tree with the goal "Remove negative reviews from the course website". (2p)

Question 8: Secure software engineering (2 points)

Why is it important to consider security during the whole software development lifecycle? Motivate!

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.