Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2010-08-27
Sal	TER1
Tid	14-18
Kurskod	TDDC90
Provkod	TEN1
Kursnamn/benämning	Programvarusäkerhet
Institution	IDA
Antal uppgifter som ingår i tentamen	10
Antal sidor på tentamen (inkl. försättsbladet)	6
Jour/Kursansvarig	Shanai Ardi/Anna Vapen
Telefon under skrivtid	282608/288986
Besöker salen ca kl.	15, 17
Kursadministratör (namn + tfnnr + mailadress)	Madeleine Häger 282360, madha@ida.liu.se
Tillåtna hjälpmedel	Inga
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

2010-08-27

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Shanai Ardi/Anna Vapen

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: Secure coding (2 points)

Assuming that integers *x* and *y* have been read from an untrusted user, give two reasons that the following code is dangerous:

```
char *p = malloc(x * y);
while (x > 0)
      while (y > 0)
            *p++ = 0;
```

Question 2: Vulnerabilities (2 points)

Briefly explain one compiler-based method for preventing exploitation stack-based overflows.

Question 3: Fuzz testing (2 points)

Explain what fuzz testing is, and describe the kinds of vulnerabilities it is good for detecting.

Question 4: SDL (2 points)

Name and briefly explain two activities in SDL that take place in the design phase

Question 5: Vulnerabilities (4 points)

Explain what a format string vulnerability is and what kind of impact(s) such vulnerabilities can have. Give an example of a vulnerable line of code, and explain how it could be exploited by an attacker.

Question 6: Threat modeling (4 points)

Explain what attack trees are and how they can be used in security. Draw an attack tree for breaking into your own apartment or dorm room. You will be judged, in part, on how complete your attack tree is.

Question 7: Static analysis (4 points)

To get precise results in static analysis, *points-to analysis* must be performed. Explain what points-to analysis is and why it is important for static analysis. Give an example in code or detailed pseudocode that illustrates a situation where points-to analysis is required. Explain your example.

Question 8: Secure design patterns (4 points)

Explain the privilege separation design pattern in detail, including the security benefits of using this pattern.

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: Common Criteria (6 points)

Explain what the Common Criteria is, and what its purpose is. Name and briefly explain all the main components and concepts of the Common Criteria.

Question 10: Vulnerabilities and detection (6 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`.

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).

Continues on next page

- 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.

Code for question 10

```
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.