Information page for written examinations at Linköping University

Examination date	2018-01-10
Room (3)	U1(36) U2(30) U3(14)
Time	8-12
Course code	TDDC90
Exam code	TEN1
Course name **Exam name**	Software Security (Software Security) Written examination (Skriftlig tentamen)
Department	IDA
Number of questions in the examination	7
Teacher responsible/contact person during the exam time	Ulf Kargén
Contact number during the exam time	013-285876
Visit to the examination room approximately	09:30, 11:00
Name and contact details to the course administrator (name + phone nr + mail)	Madeleine Häger Dahlqvist, 013-282360, madeleine.hager.dahlqvist@liu.se
Equipment permitted	Dictionary (printed, NOT electronic)
Other important information	
Number of exams in the bag	

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

Written exam

TDDC90 Software Security

2018-01-10

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Ulf Kargén, 013-285876

Instructions and grading

You may answer in Swedish or English.

There are 7 questions on the exam. Your grade will depend on the total points you score. The maximum number of points is 40. The following grading scale is preliminary and might be adjusted during grading.

Grade	3	4	5
Points required	20	29	35

Question 1: Secure software development (4 points)

a) In which phase of the security development life cycle should static analysis be used?

b) During the verification phase of SDL (the fourth phase), there are three main activities ("practices" with SDL terminology) that should be performed. Briefly explain these three activities.

Question 2: Exploits and mitigations (5 points)

a) For each of the three exploit techniques below, state whether DEP is effective in mitigating the technique. Motivate each of the answers in a sentence or two. Answers without motivation gives no points.

   i. ROP

   ii. NOP-sleds

   iii. return-to-libc

b) Address Space Layout Randomization (ASLR) is generally less effective on 32-bit systems, compared to 64-bit systems. Explain why.

Question 3: Design patterns (5 points)

Explain the following two design patterns: *secure factory* and *defer to kernel*. For each pattern your answer should include a diagram, pseudo-code and an explanation of why and when the pattern should be used.

Question 4: Web security (6 points)

a) It has been stated that disallowing GET-requests, and instead using POST, is a way to mitigate cross-site request forgery (CSRF). Is this statement correct? If yes, explain why. If no, give a counterexample of how CSRF-attacks can still be carried out.

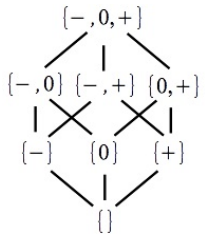
b) Explain, in a sentence or two, the difference between stored and reflected cross-site scripting (XSS).

c) Using pseudo-code, write server-side code that contains a vulnerability that allows for XSS. Your code should be detailed enough that it is clear how XSS attacks can be made. Explain your code in English (or Swedish). Give an example of a client-side request that would exploit the vulnerability in your code. Finally, suggest a modification of your code such that the vulnerability is removed. Explain why your mitigation strategy works.

Question 5: Static analysis (7 points)

The following function computes the n^{th} term of the arithmetic sum $s_n = \Sigma_{i:0 \leq i \leq n} i$. For instance $s_0 = 0$, $s_1 = 1$, $s_2 = 3$ and $s_3 = 6$. Here, `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

```
1  int sum(int n){
2    if(n <= 0)
3       return 0;
4    int s = 0;
5    while (n > 0){
6      s = s + n;
7      n = n - 1;
8      assert(n < s);
9    }
10   assert(0 <= s);
11   return s;
12 }
```



We aim to check the assertions `(n < s)` at line 8 and `(0 <= s)` at line 10. In the first part, we consider the following two approaches for checking a given assertion:

- Symbolic execution: builds a path formula obtained by violating the assertion after following a path through conditional statements (such as the one at line 2) and loops (such as the one at line 5) by choosing some outcome for the involved condition (for example, choosing $(n \leq 0)$ at line 5 in order to exit the loop and get to line 10).
- Abstract interpretation: here using the abstract values depicted in the lattice above. Intuitively, the abstract values are used to over-approximate, in an as precise manner as possible, the information of whether a variable is 0, positive, negative, or some combinations of these.

Questions:

1. Consider the assertion `(0 <= s)` at line 10:
   (a) Give a path formulas that would correspond to taking the else outcome of the if statement (line 2), entering the loop once (i.e., one iteration of the loop), exiting the loop to get to line 10 and violating the assertion there (i.e. violating the `(0 <= s)` assertion). (2 pt)
   (b) Can abstract interpretation, based on the sign abstract domain mentioned above, establish that the assertion is never violated? explain by annotating each line with the abstract element associated to each variable and obtained at the end of such an analysis. (1pt)
2. Consider the assertion `(n < s)` at line 8:
   (a) What does it mean for the predicate $wp(stmt, Q)$ to be the weakest precondition of a predicate Q with respect to a program statement $stmt$? (1 pt)
   (b) Give P_7 defined as the weakest precondition of the predicate `(n < s)` with respect to the assignment `n= n - 1` at line 7; then give P_6 defined as the weakest precondition of the predicate P_7 with respect to the assignment `s = s + n` at line 6. (2pt)
   (c) Can you argue whether P_6 is an invariant of the loop? would having P_6 as an invariant of the loop be enough to establish the assertion at line 8? justify. (1pt)

Question 6: Security testing (7 points)

a) AFL is a recent successful greybox fuzzer. Explain the main ways in which AFL's approach differs from traditional mutation-based fuzzers.

b) Mutation-based fuzzers typically does not work so well for fuzzing implementations of stateful network protocols. Explain why, and name a fuzzing technique more suited for this use case.

c) Give pseudocode for a small program with a bug that would be easy to find using concolic testing, but hard to find using mutation-based fuzzing. Explain your reasoning!

Question 7: Vulnerabilities in C/C++ programs (6 points)

The code on the next page shows part of a program that reads data from a simple data stream format. Each message in the stream consists of two 32-bit integers, followed by a variable-size data part. The first number specifies the type of data in the message, and the second number specifies the length of the data. The data part then follows directly.

The part of the program we are concerned with (`print_text_record`) parses a message, and prints its data if it is of text type (plain text or XML). We also give function declarations for a simple API for reading data from streams. It can be assumed that the API functions are correctly implemented, and function in accordance with the comments given above each function declaration. The program may be used to read streams from untrusted sources.

`print_text_record` contains at least one serious bug that can lead to a potentially exploitable condition. Explain what the bug is, and how to fix it. Clearly explain what the consequence would be of triggering the bug.

```c
/* Constants defining message data types. */
enum DStream_Types {
   DS_PLAIN_TEXT  = 1,
   DS_XML_TEXT    = 2,
   DS_RGB_RASTER  = 3
};


/* Reads 4 bytes off the stream 'ds' and returns them as an int. */
int DStream_read_int32(struct DStream* ds);


/* Reads 'sz' bytes from stream 'ds' into buffer 'dest' */
void DStream_read_data(struct DStream* ds, char* dest, size_t sz);


/* Check if a previous operation on 'ds' has caused an error.
   Returns zero (false) in case of no error, and non-zero otherwise. */
int DStream_has_error(struct DStream* ds);


int print_text_record(struct DStream* ds)
{
   char buffer[1024];

   int type = DStream_read_int32(ds);
   int size = DStream_read_int32(ds);

   // Check if we failed to read header from stream
   if(DStream_has_error(ds))
      return -1;

   size_t text_size = size;

   if(text_size <= 0)
      return -1; // Invalid size

   if(type == DS_PLAIN_TEXT)
      printf("Type: TEXT\n");
   else if(type == DS_XML_TEXT)
      printf("Type: XML\n");
   else
      return 1; // Not a text-type record

   // Check that input fits in buffer
   if(size >= 1024)
      return -1; // Oversized record

   // Read into buffer
   DStream_read_data(ds, buffer, size);

   if(DStream_has_error(ds))
      return -1; // Error reading string off stream

   buffer[size] = 0; // Make sure string is terminated

   // Print string
   printf("%s\n", buffer);

   return 0; // Signal success
}
```