Information page for written examinations at Linköping University

Examination date	2016-08-24
Room (1)	TER4
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	09:00, 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

2016-08-24

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) During which phase or phases of SDL are bug bars evaluated?

b) Explain by example how attack trees are created (come up with a scenario on your own, and make sure that you explain all the details of attack trees).

Question 2: Exploits and mitigations (5 points)

a) Clearly explain the purpose of using a NOP-sled when exploiting a stack based buffer overflow.

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 privilege separation. 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)

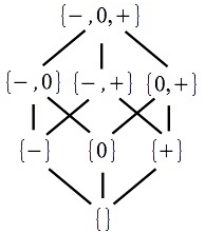
The developers of a website are considering adding functionality that would allow users to upload files to the server, that later will be downloaded and used by other users. There are several vulnerabilities that could be introduced by allowing this, please explain five of these. Your answer should include an explanation of the vulnerabilities, the possible consequences and how they can be mitigated.

Question 5: Static analysis (7 points)

The following function computes the n^{th} element of the Fibonacci sequence "0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..." (i.e., the sequence starting with $a_0 = 0$, $a_1 = 1$ and $a_n = a_{n-1} + a_{n-2}$ for $n \geq 2$).

Here, int denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

```
1  int fib(int n){
2    if(n <= 0)
3       return 0;
4    int tmp = 0;
5    int a1  = 1;
6    int a2  = 0;
7    while (n > 0){
8      tmp = a1 + a2;
9      a2 = a1;
10     a1 = tmp;
11     assert(a2 <= a1);
12     n = n - 1;
13   }
14   assert(0 <= a1);
15   return a1;
16 }
```



We aim to check the assertions `(a2 <= a1)` at line 11 and `(0 <= a1)` at line 14. 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 7) by choosing some outcome for the involved condition (for example, choosing $(n \leq 0)$ at line 7 in order to exit the loop and get to line 14).
- 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 <= a1)` at line 14:
   (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 14 and violating the assertion there (i.e. violating the `(0 <= a1)` 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 obtained at the end of such an analysis. (1pt)
2. Consider the assertion `(a2 <= a1)` at line 11:
   (a) Give P_{10} defined as the weakest precondition of the predicate `(a2 <= a1)` with respect to the assignment `a1 = tmp` at line 10; then give P_9 defined as the weakest precondition of the predicate P_{10} with respect to the assignment `a2 = a1` at line 9; finally give P_8 defined as the weakest precondition of the predicate P_9 with respect to the assignment `tmp = a1 + a2` at line 8. (2pt)
   (b) Suppose the program variables satisfy P_8 just before the assigment `tmp = a1 + a2` at line 8. If the program continues for three steps, can it violate the assertion `(a2 <= a1)` at line 11? (1pt)
   (c) What should `a2` satisfy at the entrance of the loop at line 8 in order for P_8 to always hold? (1pt)

Question 6: Security testing (7 points)

a) Briefly explain two general reasons (i.e. not related to specific vulnerability types) why automated fuzzing of web applications is often harder than fuzzing e.g. a desktop program written in C.

b) Consider cross-site scripting (XSS) vulnerabilities. Which of the two vulnerability types *Stored XSS* and *Reflected XSS* is generally easier to detect using a black-box web application fuzzer? Clearly motivate your answer.

c) Explain why detecting cross site request forgery (CSRF) bugs using automated testing is often very difficult.

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

The code on the next page shows the beginning of a function that receives and processes a video stream from a potentially untrusted source over a network (e.g. the internet). The specific nature of the processing of video streams is not important here. The function contains at least one serious security bug.

a) Identify and name the vulnerability. Clearly explain how an attacker could trigger the bug.

b) Explain how to fix the bug.

```
#define BUF_SIZE 2000000

// Represents a video stream. Details unimportant here.
struct Stream;

enum {
   QUALITY_LOW  = 1,
   QUALITY_MED  = 2,
   QUALITY_HIGH = 3
};

// Receive maximum 'size' bytes from stream 's' into memory pointed to
// by 'dst'. Returns the number of bytes actually read from stream. (Can be
// lower than 'size' if more data than what was available was requested.)
size_t receive(const struct Stream* s, size_t size, void* dst);

// Receives a video stream and processes it.
// Returns -1 in case of error, 0 otherwise.
int handleStream(const struct Stream* s)
{
   char buffer[BUF_SIZE];

   int quality;
   size_t n_seconds;
   size_t n_received;
   size_t data_rate;

   // Read header fields from stream:

   // First quality setting ...
   n_received = receive(s, sizeof(quality), &quality);
   if(n_received != sizeof(quality)) {
      printf("Transmission error!\n");
      return -1;
   }

   // ... and then number of seconds in stream
   n_received = receive(s, sizeof(n_seconds), &n_seconds);
   if(n_received != sizeof(n_seconds)) {
      printf("Transmission error!\n");
      return -1;
   }

   switch(quality) {
      case QUALITY_LOW:
         data_rate = 8192;
         break;
      case QUALITY_MED:
         data_rate = 16384;
         break;
      case QUALITY_HIGH:
         data_rate = 32768;
         break;
      default:
         printf("Unknown quality setting!\n");
         return -1;
   }

   if(n_seconds*data_rate > BUF_SIZE) {
      printf("Too much data!\n");
      return -1;
   }

   // Recieve stream data into 'buffer', one second at a time
   for(size_t i = 0; i < n_seconds; i++) {
      n_received = receive(s, data_rate, buffer + i*data_rate);
      if(n_received != data_rate) {
         printf("Transmission error!\n");
         return -1;
      }
   }

   // Continue processing data...
```