Information page for written examinations at Linköping University

Examination date	2019-04-26
Room (1)	TER4(12)
Time	8-12
Edu. code	TDDC90
Module	TEN1
Edu. code name **Module 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

2019-04-26

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) Consider the general software development lifecycle. In order to secure the lifecycle we can introduce security touch points. Draw the lifecycle and annotate where in the cycle you would use: *misuse cases*, *static analysis* and *penetration testing*.

b) Name Phase 2 of Microsoft's SDL (not counting the pre-SDL phase) and briefly explain the activities prescribed by SDL in this phase.

Question 2: Exploits and mitigations (5 points)

It is mentioned during the lectures that ASLR and DEP (non-executable memory) should be used together to be effective at mitigating exploits. Name and briefly explain an exploit method that is possible if

a) ASLR is used but not DEP

b) DEP is used but not ASLR

For each of the two cases, also explain why the exploit method would fail if both mitigations were used.

Question 3: Design patterns (5 points)

a) Explain the design pattern *secure factory*. Your answer should include a diagram, pseudo-code and an explanation of why and when the pattern should be used.

b) Both of the patterns *privilege separation* and *defer to kernel* are specialized versions of a more general design pattern. Give the name of this general pattern, and explain the intent and motivation of it in 2-3 sentences.

Question 4: Web security (6 points)

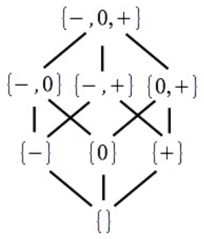
a) Explain, in a sentence or two, the difference between stored and reflected cross-site scripting (XSS).

b) 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 *three* 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, for a positive n, the sum of all even numbers smaller or equal to n, i.e., $\texttt{evenSum(n)} = 2 + 4 \ldots + 2 * p$ with $n = 2 * p$ or $n = 2 * p + 1$. Here, `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

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



We aim to check the assertions `(s == i * (i-1))` at line 7 and `(1 <= i)` at line 9.
Questions:

1. Consider first the assertion `(1 <= i)` at line 9:
   (a) Symbolic execution: Give a path formulas that would correspond to entering the loop once (i.e., one iteration of the loop), exiting the loop to get to line 9 and violating the assertion there (i.e. violating the `(1 <= i)` assertion). (2 pt)
   (b) Abstract interpretation: can an abstract interpretation analysis 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 (i.e., each one of `i, s` and `n`) and obtained at the end of such an analysis (i.e., after the analysis converges). (1pt)

2. Consider now the assertion `(s == i*(i-1))` at line 7:
   (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_6 defined as the weakest precondition of the predicate `(p == i*(i-1))` with respect to the assignment `i = i + 1` at line 6; then give P_5 defined as the weakest precondition of the predicate P_6 with respect to the assignment `s = s + 2*i` at line 5. (2pt)
   (c) Is P_5 an invariant of the loop? would having P_5 as an invariant of the loop be enough to establish the assertion at line 7? justify. (1pt)

Question 6: Security testing (6 points)

a) A generation based fuzzer generally requires two components to work: A grammar and a set of fuzzing heuristics. Explain the purpose of both these components.

b) Greybox fuzzing is sometimes also called *evolutionary fuzzing*. Explain why.

c) Briefly explain two reasons why automated fuzzing of web applications is often harder than fuzzing e.g. a desktop program written in C.

Question 7: Vulnerabilities in C/C++ programs (7 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 *two* different (but related) security vulnerabilities.

a) For *each* of the two vulnerabilities, identify the bug in the code, name the vulnerability type, and explain how an attacker could use the bug to hijack the control flow of the program. (You don’t need to explain how to craft a full exploit.)

b) Explain how to fix each bug.

```c
#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;
   int n_seconds;
   int n_received;
   int 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'
   n_received = receive(s, n_seconds * data_rate, buffer);
   if(n_received != n_seconds * data_rate) {
      printf("Transmission error!\n");
      return -1;
   }

   // Continue processing data...
```