Information page for written examinations at Linköping University

Examination date	2018-04-06
Room (1)	TER1(30)
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

2018-04-06

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) How is probability and consequence represented in CORAS, and how are risks compared? Use a small example to illustrate your explanation.

b) In Security Development Lifecycle (SDL), how are bug bars and quality gates different?

Question 2: Exploits and mitigations (5 points)

For each of the two mitigations below, give a high-level explanation of how it works. Also, for each of the two, name and explain one exploit technique that is specifically designed to overcome the mitigation.

a) DEP

b) ASLR

Question 3: Design patterns (5 points)

Explain the following two design patterns: *privilege separation* and *secure logger*. 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)

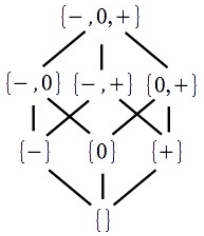
a) Using pseudo-code, write server-side code that contains a vulnerability that allows for SQL injections. Your code should be detailed enough that it is clear how SQL injections can be made. Explain your code in English. Give an example of a client-side request that would exploit the vulnerability in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

b) Explain how a slow HTTP-POST attack works. What is an attacker attempting to achieve by executing such an attack? How can the risk of such an attack be mitigated?

Question 5: Static analysis (7 points)

The following function takes two integers `n` and `m` as input. It computes the remainder of the euclidean division of `n` by `m` in case both of them are strictly positive. It otherwise returns `-1`. For instance `remainder 10 2` gives 0 because 10=5*2+0 while `remainder 20 3` gives 2 because 20=3*6+2. Here, `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

```
1  int remainder(int n, int m){
2    if(n <= 0 || m <= 0)
3       return -1;
4    int r = n;
5    int q = 0;
6    while (r >= m){
7      r = r - m;
8      q = q + 1;
9      assert(n = m * q + r);
10   }
11   assert(0 <= r);
12   return r;
13 }
```



We aim to check the assertions `(n = m * q + r)` at line 9 and `(0 <= r)` at line 11. 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 6) by choosing some outcome for the involved condition (for example, choosing $(r < m)$ at line 6 in order to exit the loop and get to line 11).
- 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 <= r)` at line 11:
   (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 11 and violating the assertion there (i.e. violating the `(0 <= r)` 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 = m * q + r)` at line 9:
   (a) Give P_8 defined as the weakest precondition of the predicate `(n = m * q + r)` with respect to the assignment `q= q + 1` at line 8; then give P_7 defined as the weakest precondition of the predicate P_8 with respect to the assignment `r = r - m` at line 7. (2pt)
   (b) Argue whether P_7 is an invariant of the loop? would having P_7 as an invariant of the loop be enough to establish the assertion at line 9? Justify. (2pt)

Question 6: Security testing (7 points)

a) What is the benefit of using a dynamic-analysis tool such as Valgrind or AddressSanitizer during fuzzing? Give an example! What is the downside?

b) What is the problem with *magic constants* in fuzzing? Which of blackbox, greybox, or whitebox fuzzing is best suited to deal with this problem? Briefly motivate.

c) In the context of mutation-based fuzzing, what is a seed input?

d) Imagine that you are tasked with performing mutation-based fuzzing on an image viewer for JPEG images. In order to thoroughly test the program, the fuzzer should be run with several different seed inputs. You realize that you have a bunch of vacation photos from last summer that you can use as seeds. If you want to maximize the chance of finding bugs, what may be the problem with this seed-selection approach? Also suggest a better approach.

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

The code on the next page shows the beginning of a program that performs some privileged operation, the nature of which is not important here. The program makes sure that the user is allowed to perform the privileged operation by first asking for a password and user name, and checking these against a user database. (Exactly how the database works is also not important here.) The program has an implementation error, which could potentially be exploited by attackers.

a) Identify and name the vulnerability. Explain how an attacker could trigger the bug. You *don't* need to explain how to exploit the bug.

b) Explain how to fix the bug.

Note: It is best-practice to clear (overwrite) passwords stored in memory as soon as possible, to minimize the chance that someone can get them by e.g. dumping RAM contents. We here assume that this is not a possible attack scenario on the system that this code runs on.

```c
// Read input string from user and store in 'dst'. A maximum
// of 'max_size' bytes will be written to 'dst', including the
// NULL terminator. Returns 0 if successful, or -1 if the
// input does not fit in 'dst'.
int get_input(char* dst, size_t max_size);

// Returns the user ID for the given user name, or -1 if the
// user name is unknown.
int get_user_id(const char* username);

// Fetches the password for given user ID from the database
// and stores it in 'dst'. Passwords cannot be longer than 20
// characters. A maximum of 21 bytes can thus be written to
// 'dst'. Returns 0 if successful, or -1 if the operation
// failed due to an invalid user ID.
int get_password(char* dst, int user_id);

int main(void) {
   const size_t BUF_SIZE = 25;
   int id = -1;
   int failed = 0;

   printf("Username: ");
   char* user = malloc(BUF_SIZE);
   if(get_input(user, BUF_SIZE) != 0) {
      // Too long input
      printf("Error: Too long username!\n");
      free(user);
      failed = 1;
   } else {
      id = get_user_id(user);
   }

   printf("Password: ");
   char* given_password = malloc(BUF_SIZE);
   if(get_input(given_password, BUF_SIZE) != 0) {
      // Too long input
      printf("Error: Too long password!\n");
      free(user);
      free(given_password);
      failed = 1;
   }

   if (failed != 1) {
      char* real_password = malloc(BUF_SIZE);
      if(get_password(real_password, id) != 0) {
         printf("Invalid user name!\n");
         failed = 1;
      } else if(strcmp(real_password, given_password) != 0) {
         // If real password is not identical to given password,
         // authentication fails
         failed = 1;
      }
      free(user);
      free(given_password);
      free(real_password);
   }

   if(failed == 0)
      printf("Authenticaton succeeded!\n");
   else {
      printf("Authenticaton failed!\n");
   }

   if(failed == 1) {
      exit(2); // Quit program if autentication failed
   }

   // Continue processing...
```