Information page for written examinations at Linköping University

Examination date	2016-01-13
Room (1)	KÅRA
Time	14-18
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	15:00, 17:00
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

2016-01-13

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 the implementation phase of SDL (the third phase), there are three main activities that should be performed. Explain these three activities

b) In which phase of the security development life cycle should misuse cases be used?

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. Explain why this is so. In your answer, make sure to give examples of why neither mitigation provides sufficient protection on its own.

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)

a) Describe the rule-based version of a brute force attack.

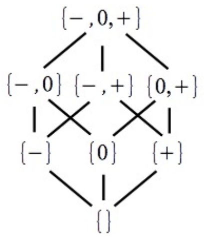
b) Describe two different vulnerabilities that can result from allowing users to upload files, and explain how these vulnerabilities can be mitigated.

c) Explain, using a combination of pseudo-code and English, why email clients should block images from untrusted sources. Your answer should be a complete example of how an attacker could use images to stage a cross-site request forgery.

Question 5: Static analysis (7 points)

Consider the following `foo` function, where int denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

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



We aim to check the assertions `((r + 2*n) == 2*m)` at line 10 and `(n <= 0)` at line 12. Given an assertion, we consider the following two approaches:

- Symbolic execution: builds a path formula obtained by violating the assertion after unrolling the loop at line 4 k times, choosing some outcomes for the if statement at line 5.
- 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. Which of the two above approaches is sound? what does it mean? (1pt)
2. Consider the assertion `(n <= 0)`:
   (a) Give two path formulas (corresponding to two outcomes of the if statement at line 5) corresponding to violating the assertion after one complete iteration of the loop. Is any of the formulas satisfiable? (1pt)
   (b) Suppose `n` is positive, how many such paths are there (as a function of `n`)? Can symbolic execution exclude violation of the assertion irrespective of the input (i.e. for all possible inputs) after running it at "compile time"? (1pt)
   (c) 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)
3. Consider the assertion `((r + 2*n) == 2*m)`:
   (a) Give `P`, the weakest precondition of the predicate `((r + 2*n) == 2*m)` with respect to the assignment `n = n - 1` at line 9? (1pt)
   (b) Give `Q`, the weakest precondition of the predicate `P` with respect to the assignment `r = r + 2` at line 8? (1pt)
   (c) Can abstract interpretation, based on the sign abstract domain mentioned above, establish the assertion is never violated? explain by using the annotations you used in question "2.(c)" above. (1pt)

Question 6: Security testing (7 points)

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

b) Generation-based fuzzers do typically not need seed inputs. Explain what purpose the seed inputs fulfil in mutation-based fuzzing, and why this is not needed in a generation-based fuzzer.

c) 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.

d) Whitebox fuzzing also makes use of seed inputs. What is the purpose of the seeds in this context?

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 a very serious implementation error, which could allow attackers to gain unauthorized access.

a) Identify and name the vulnerability. Explain how an attacker can exploit the vulnerability to gain unauthorized access. You may need to make some assumptions about how specific system components work. Clearly state these!

b) Explain how to fix the bug.

```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 = 256;
   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);
      exit(1); // Quit program
   } 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);
      failed = 1;
   }

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

   if(failed == 0)
      printf("Authenticaton succeeded for user %s!\n", user);
   else {
      printf("Authenticaton failed for user %s!\n", user);
   }

   // Zero out password in memory
   memset(real_password, 0, BUF_SIZE);

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

   // Continue processing...
```