Information page for written examinations at Linköping University

Examination date	2020-01-15
Room (1)	TER4(41)
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)	Annelie Almquist, 013-282934, annelie.almquist@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
Ulf Kargén

Written exam

TDDC90 Software Security

2020-01-15

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

Question 2: Exploits and mitigations (5 points)

a) Give a high-level description (a few sentences) of how *Control-Flow Integrity* (CFI) works.

b) Name one attack technique that is prevented by CFI but not by DEP. Explain why CFI is effective for stopping the attack and why DEP isn't.

Question 3: Design patterns (5 points)

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

b) Explain the intent and motivation of the *Resource Acquisition Is Initialization* (RAII) pattern. Name a type of vulnerability that is avoided by using this pattern.

Question 4: Web security (6 points)

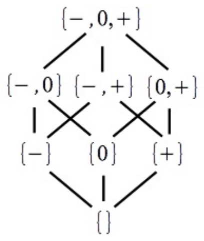
a) Explain, in a few sentences, the difference between stored and reflected cross-site scripting (XSS).

b) Use pseudo-code and English to explain what an Insecure Deserialization vulnerability is, and how it can be exploited by attackers.

Question 5: Static analysis (7 points)

The following function computes the cubic product $n * n * n$ of a natural number n. Here, `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

```
 1 int foo(int n){
 2   if( (n <= 0))
 3      return 0;
 4   int a = 1;
 5   int b = 1;
 6   int c = 1;
 7   while (a < n){
 8     a = a + 1;
 9     b = b + 2*a - 1;
10     c = c + 3*b - 3*a + 1;
11     assert(c == a * a * a);
12   }
13   assert(c > 0);
14   return c;
15 }
```



We aim to check the assertions `(c == a * a * a)` at line 11 and `(c > 0)` at line 13. Questions:

1. Consider first the assertion `(c > 0)` at line 13:

   (a) Symbolic execution: 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 13 and violating the assertion there (i.e. violating the `(c > 0)` 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 `a, b, c` and `n`) and obtained at the end of such an analysis (i.e., after the analysis converges). (1pt)

2. Consider now the assertion `(c == a * a * a)` at line 11:

   (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_8 defined as the weakest precondition of the predicate `(c = a * a * a)` with respect to the sequence of assignments at lines 8, 9 and 10 (2pt)

   (c) Is P_8 an invariant of the loop? Argue (1pt)

Question 6: Security testing (7 points)

a) Greybox fuzzing is often exponentially faster at finding bugs than a simple mutational black-box fuzzer. Explain why, using a small example.

b) What is the path explosion problem in concolic testing?

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 next page shows code for a simple utility `add_user`, which works similar to the Unix `useradd` command, but which is intended for situations where also non-root users must be able to add new accounts to the system. Since the tool must be able to change the password file, among other things, it is configured as SUID root, meaning that it can perform operations requiring root privileges even if it is invoked by a regular user. The tool takes the name of the new account as a mandatory argument. By default, the tool creates a regular user account. An optional argument “`--root`” can also be supplied for creating a new account with root privileges. This option, however, is restricted to work only if a root user is invoking the tool, to prevent privilege escalation.

The program contains an implementation error, which constitutes a serious security vulnerability.

a) Identify the bug in the code and name the vulnerability type.

b) Explain how an attacker could exploit the bug. Your answer should show in detail what the input to the program must look like in order to succeed with the exploit, and what the exploit leads to. State any assumptions about e.g. memory layout that you make.

c) Explain how to fix the bug.

```c
/* Add regular user account with provided username.
   Queries for password for new account when called.
   Details unimportant. */
void add_account(const char* username);

/* Add root-privilege account with provided username.
   Queries for password for new account when called.
   Details unimportant. */
void add_root_account(const char* username);

int main(int argc, char** argv)
{
    size_t idx;
    int restrict_root = 1;
    char username[32];

    // Remove restrictions if root user is invoking program, indicated by
    // a real user ID of 0. (The effective user ID is always 0 since the
    // the program is SUID root.)
    if(getuid() /* get real UID */ == 0)
        restrict_root = 0;

    // Check argument count.
    // First argument (argv[0]) is always name of executable,
    // second argument (argv[1]) should be username
    if(argc < 2) {
        printf("Too few arguments. Usage: add_user USERNAME [--root]\n");
        exit(1); // Exit with error code
    }

    if(strlen(argv[1]) > 32) {
        printf("Usernames must be at most 32 letters\n");
        exit(1);
    }

    strcpy(username, argv[1]);

    // Username must only contain lowercase characters,
    // according to our system policy
    idx = 0;
    while(username[idx] != 0) {
        username[idx] = tolower(username[idx]);
        idx++;
    }

    // Check for third optional "--root" argument
    if(argc > 2 &&
       strcmp("--root", argv[2]) == 0
       /* strcmp returns 0 if strings are equal */)
    {
        if(restrict_root) {
            printf("Access is restricted! Are you logged in as root?\n");
            exit(1);
        }
        add_root_account(username);
    } else {
        add_account(username);
    }

    return 0;
}
```