LiTH, Linköpings tekniska högskola
IDA, Institutionen för datavetenskap
Ulf Kargén

Written exam

TDDC90 Software Security

2025-08-27

Permissible aids

Dictionary (printed, NOT electronic)

Teacher on duty

Ulf Kargén, 013-285876

Instructions and grading

There are 7 questions on the exam. Your grade will depend on the total points you score. The maximum number of points is 38. The following grading scale is preliminary and might be adjusted during grading. You may answer in Swedish or English.

Grade	3	4	5
Points required	19	27	32

Question 1: Secure software development (4 points)

Name phases *2 and 3* of Microsoft's SDL and briefly explain the activities prescribed in each of these phases.

Question 2: Memory safety (5 points)

Consider the two attack methods below, each designed to overcome a specific exploit mitigation. For each of the two methods, give a high-level explanation of the exploit mitigation technique it was designed to circumvent, and how the attack is able to circumvent that mitigation

a) Return-to-libc

a) Heap spraying

Question 3: Design patterns (4 points)

a) Come up with a (realistic) example of a vulnerability and corresponding attack that is made possible by failing to adhere to the *secure logger* pattern.

b) Explain the design pattern *secure factory*, including an explanation of why and when the pattern should be used.

Question 4: Web security (6 points)

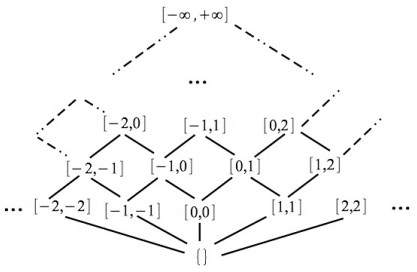
a) Explain using examples how (1) *SQL injection* and (2) *command injection* attacks work.

b) Both of the above attacks are made possible by a similar implementation flaw. Clearly describe it.

Question 5: Static analysis (7 points)

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

```
1  int foo(int n){
2    int rslt = 0;
3    int i = n;
4    while(i != 0){
5      if(n > 0){
6        rslt = rslt + 2;
7        i = i - 1;
8      }else{
9        rslt = rslt - 2;
10       i = i + 1;
11     }
12   }
13   assert(i == 0);
14   assert(rslt == 2*n);
15   return rslt;
16 }
```



Questions:

1. Symbolic execution:

   - How many possible (i.e. feasible) paths does the method `foo` have if the variable `n` can assume any one of the 11 values in `[-5,5]`? (1pt)
   - Give, without discussing its possible satisfiability, the SSA formula for the path condition that executes executes once `rslt = rslt + 2` (i.e., passes once through line 6), does not execute `rst = rslt - 2` (i.e., does not pass through line 9), respects the assertion at line 13 (i.e., `i == 0`) and violates the assertion at line 14 (i.e., `rslt == 2*n`). Observe there is one path condition following this description. It might not be satisfiable, but it should follow the control flow of the program. (2 pts)

2. Abstract interpretation: Annotate, after convergence of a most precise analysis, the start of each line in `foo` with an abstract element associated to each one of the defined variables (including the unknown value passed as parameter to `foo`). For each line, and for each variable or parameter, give the interval domain obtained at the end of the analysis. Interval domains are elements of the lattice to the right of `foo`. Observe this is not the sign domain discussed in the course. (2pt)

3. Give, without justification, the weakest condition P such that executing the sequence of three assignments `z= x + 2*y; x= z - x; y= 2*(z - x);` (where `x`, `y` and `z` are integer variables with no integer overflows) from a configuration satisfying P will always result in values satisfying the condition ($Q : x = y$ and $z \leq x$). In other words, give the weakest (i.e., most general) predicate P such that the Hoare triple $\{P\}$`z= x + 2*y; x= z - x; y= 2*(z - x);`$\{Q\}$ holds (i.e., is always true or valid). (2pt).

Question 6: Security testing (6 points)

a) Imagine that you are tasked with creating a black box fuzzer for a JavaScript interpreter. Which fuzzing strategy (mutation or generation) would be most appropriate? Clearly motivate your answer.

b) Explain what *in-memory fuzzing* is, and why it is used.

c) Give pseudocode for a small program with a bug that would be found (relatively) quickly using greybox fuzzing, but would take a long time to find using black-box mutational fuzzing. Clearly explain why your code has this property.

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

The code on the next page shows a function `login` for handling password checks as part of a login system. The function first queries for the user name, and then gives the user the option to either type in the password, cancel the login, or request that a password reset link be sent to a provided email address (given that the provided email address matches with the one associated with the username). The code contains a serious security bug. (We here ignore the other glaring security issue that passwords are stored in plain text.)

a) Identify the bug in the code.

b) Clearly explain with an example how an attacker could exploit the bug, and what the consequences of a successful exploit would be. State any (reasonable) assumptions you make about, e.g., memory layout.

c) Explain how to fix the bug.

You can assume that all code comments are truthful and correct.

```c
#define USR_LEN 16
#define PWD_LEN 16
#define EMAIL_LEN 32

// Fetches password for given username and stores it into 'destination'
// Passwords are never longer than 15 characters.
// Returns 0 on success and 1 on error.
int get_pwd_from_user_db(const char* username, char* destination);

// Sends a password-reset link to given email, provided that the email is
// registered with the given username. Returns 0 on success and 1 on error.
int request_email(const char* username, const char* email);

// Performs credentials check. Returns 0 on successful login and 1 on
// failed login or error.
int login()
{
   char username[USR_LEN];
   char given_password[PWD_LEN];
   char email[USR_LEN];
   char stored_password[PWD_LEN];
   int choice;

   printf("User name: ");
   // Read string from standard input. Given format specifier "%Ns" fscanf will
   // read at most N characters into destination buffer ('username' here), or
   // until whitespace (e.g. newline) is encountered. A null terminator is
   // always appended after the last read character in the destination buffer.
   fscanf(stdin, "%15s", username);
   if(get_pwd_from_user_db(username, stored_password) != 0) {
      printf("Error accessing user database\n");
      return 1;
   }

   // Loop until explicitly terminated with return statement
   while(1) {
      printf("Choose action:\n");
      printf("   1: Proceed to type password\n");
      printf("   2: Cancel login\n");
      printf("   3: Request password reset email\n");
      // Read one integer from standard input into 'choice'
      fscanf(stdin, "%d", &choice);

      if(choice == 1) {
         printf("Password: ");
         fscanf(stdin, "%15s", given_password);
         // strcmp returns 0 if strings are equal
         if(strcmp(given_password, stored_password) == 0) {
            printf("Login successful!\n");
            return 0;
         } else {
            printf("Incorrect password!\n");
            return 1;
         }
      } else if(choice == 2) {
         return 1;
      } else if(choice == 3) {
         printf("Email address: ");
         fscanf(stdin, "%31s", email);
         if(request_email(username, email) != 0) {
            printf("Error sending email\n");
            return 1;
         } else {
            printf("A password reset link has been sent!\n");
         }
      } else {
         printf("Invalid choice, please try again.\n");
      }
   }
}
```