LiTH, Linköpings tekniska högskola
IDA, Institutionen för datavetenskap
Nahid Shahmehri

Written exam

TDDC90 Software Security

2017-01-14

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) Excluding the pre-SDL and final security review phases, SDL consists of five phases. Name the *second* and *third* phase, and give a brief description of them (max 60 words for each phase).

b) Map each of the following activities to the most appropriate phase of SDL: *fuzz testing* and *static analysis*.

Question 2: Exploits and mitigations (5 points)

a) A use-after-free vulnerability can sometimes be used by an attacker to redirect the flow of execution to an arbitrary address. Explain how, using an example. (You don't need to explain the whole process of achieving arbitrary code execution, just how to redirect the execution to an address of the attackers choosing.)

b) Which of the two following mitigations would provide the biggest obstacle for an attacker trying to exploit a use-after-free bug? Briefly explain why.

   i. ASLR

   ii. Stack Cookies

Question 3: Design patterns (5 points)

Explain the following two design patterns: *secure factory* and *clear sensitive information*. 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)

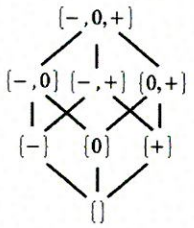
a) Using pseudo-code, write server-side code that contains a vulnerability that allows for SQL injections. Your code should be detailed enough that it is clear how SQL injections can be made. Explain your code in English. Give an example of a client side request that would exploit the vulnerability in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

b) At least one of the (web) vulnerabilities discussed in this course can be used to stage a denial of service attack. Explain in detail how this can be achieved (using pseudo-code and English). Your response should clearly explain the vulnerability, the conditions for the attack to succeed and how the attack is performed.

Question 5: Static analysis (7 points)

The following function computes the n^{th} positive multiple of 3. Here, int denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

```
1  int mult_by_3(int n){
2    if(n < 0)
3      return -1;
4    int rstl = 0;
5    int i = n;
6    while (i != 0){
7      rslt = rslt + 3;
8      i = i - 1;
9    }
10   assert(i == 0);
11   assert(rslt == 3*n);
12   return rslt;
13 }
```



We aim to check the assertions `(i == 0)` at line 10 and `(rslt == 3*n)` 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 conditions.
- 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 `(i == 0)` at line 10:
   (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 10 and violating the assertion there (i.e. violating the `(i == 0)` assertion). (2 pt)
   (b) Can abstract interpretation, based on the sign abstract domain mentioned above, establish that the assertion `(i == 0)` is never violated? explain by annotating each line with the abstract element obtained at the end of such an analysis. (2pt)
2. Consider the assertion `(rslt == 3*n)` at line 11:
   (a) Give the predicate P_7 defined as the weakest precondition of the predicate `Inv = ((rslt == 3* (n-i)) && (0 ≤ i) && (i ≤ n))` with respect to the assignment `i= i - 1` at line 8; then give P_6 defined as the weakest precondition of the predicate P_7 with respect to the assignment `rslt= rslt + 3` at line 7? (2pt)
   (b) `Inv` is in fact an invariant of the loop (lines 6-9). Using this fact, can you argue why the assertion `(rslt == 3*n)` at line 11 holds? (1pt)

Question 6: Security testing (7 points)

a) Describe the main components in a typical fuzzing framework. **Use one or two sentences per component.** Overly long answers may lead to a reduction of points!

b) Contrast *generation based* and *mutation based* fuzzing, and briefly describe their main strengths and weaknesses.

c) Depth-first-search and breadth-first-search are two simple search strategies in concolic testing. Neither of them work that well in practice on large real-life programs. Explain why.

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

The code on the next page shows a simple program that prints a list of floating-point values, and allows the user to specify a few command-line options to control how printed values are formatted. The program has the following command-line options:

```
-f INPUT        INPUT specifies name of input file. Mandatory.
-e              If this flag is given, use scientific notation when printing values.
-p PREC         PREC specifies the precision as the number of digits to print after the decimal
                point. PREC must be an integer value. Default: 8
```

For example, given the options `-f values.dat -p 3` the values in "values.dat" would be formatted something like this:

```
1: 1.234
2: 0.567
3: 89.000
...
```

The program contains at least one serious vulnerability.

a) Identify and name the vulnerability.

b) Give an example of an input that would demonstrate the vulnerable behaviour.

c) Explain (using English and/or pseudo code) how to fix the bug.

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

```c
// Omitted #include:s to save space

#define FORMAT_MAX 100
#define VALUES_MAX 1000

int use_exp = 0;
const char* precision = "8";
FILE* file = NULL;

// Prints given error message and terminates program
void exit_error(const char* message) {
   printf("%s", message);
   exit(1);
}

// Parses command line options. 'opt' is the option character, and, for options
// taking arguments, 'data' is guaranteed to point to a valid C-string. This
// string pointer is also guaranteed to be valid during the entire program execution.
// For example, given the command line argument -x ABC, we will have opt='x' and data="ABC".
void parse_options(char opt, const char* data) {
   switch (opt) {
      case 'e':
         use_exp = 1;
         break;
      case 'p':
         precision = data;
         if(precision[0] < '0' || precision[0] > '9')
            exit_error("Error: non-numeric precision option.\n");
         break;
      case 'f':
         file = fopen(data, "r"); // Open file for reading
         if(file == NULL)
            exit_error("Error opening file.\n");
         break;
      default:
         exit_error("Error: incorrect commandline arguments.\n");
   }
}

// Reads a maximum of 'max_values' float values from file stream 'f' into buffer 'destination'.
// Directly terminates program on any error. Returns number of values actually read.
// (Implementation details are unimportant.)
size_t read_values(FILE* f, float* destination, size_t max_values);

int main(int argc, char** argv)
{
   // Prefix of format specification.
   // %d prints an integer argument. %.Xf prints a floating-point argument,
   // using X digits after the decimal point, e.g. "%.8f".
   const char* prefix = "%d: %.";
   const size_t prefix_len = 6;
   const size_t suffix_len = 2;

   char format[FORMAT_MAX];

   float values[VALUES_MAX];
   size_t n_values;

   size_t i;
   char c;

   /*** Retrieve command line arguments and send to parse_options. ***/
   /*** This code can be assumed to be correct and safe.           ***/
   while ((c = getopt (argc, argv, "p:f:e")) != -1) {
      parse_options(c, optarg);
   }
   /****************************************************************/
   if(file == NULL)
      exit_error("Error: no input file specified.\n");

   if(strlen(precision) > FORMAT_MAX - prefix_len - suffix_len - 1)
      exit_error("Error: too long format specification.\n");

   // Copy prefix into 'format'
   strcpy(format, prefix);

   // Concatenate contents of 'precision' after prefix
   strcat(format, precision);

   // Finally, concatenate suffix to end of string
   if(use_exp)
     strcat(format, "e\n"); // %e formats using scientific notation
   else
     strcat(format, "f\n"); // %f formats as regular floating point

   n_values = read_values(file, values, VALUES_MAX);

   for(i = 0; i < n_values; i++) {
      printf(format, (int)i, values[i]);
   }

   return 0;
}
```