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

Written exam

TDDC90 Software Security

2024-08-28

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 (3 points)

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

Question 2: Exploits and mitigations (6 points)

a) Explain how *heap spraying* works, and which mitigation it is designed to circumvent.

b) For each of the two mitigations below, clearly explain why it can make *return-oriented programming (ROP)* attacks harder.

   a. Control-flow integrity (CFI)

   b. Address-space layout randomization (ASLR)

Question 3: Design patterns (4 points)

a) One of the design patterns mentioned in the course literature can reduce the risk of introducing use-after-free (UAF) bugs. Name this design pattern and explain the main idea of it, and why it reduces the risk of having UAF bugs in your code.

b) Explain the motivation for the secure design pattern *secure logger*. Give an example of a consequence of not using this pattern.

Question 4: Web security (6 points)

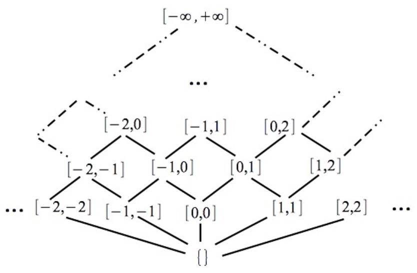
a) Explain, using an example, how an XXE vulnerability could be exploited to perform an SSRF attack. Your example should be detailed enough, so that it is possible to understand every principal step of the attack.

b) Explain how CSRF-tokens work. Using pseudocode and English/Swedish, also explain how CSRF-tokens can be implemented in a web app. You can use pseudocode resembling PHP, or any other server-side language you are familiar with, as long as it is clear how the (pseudo)code works.

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 a){
2    int rslt = 25;
3    int i = 0;
4    while(i < a){
5      rslt = rslt + 2;
6      i = i + 1;
7    }
8    assert(rslt >= a);
9    return out;
10 }
```



Questions:

1. Symbolic execution:
   - Suppose the assertion at line 8 is removed (i.e., commented out). How many possible paths does the method `foo` have if the variable `a` is known to be in `[-25,25]`? (1 pt)
   - Now suppose the assertion is put back at line 8 (i.e., it is not commented out anymore). Give, without discussing its possible satisfiability, the SSA formula for one path condition that violates the assertion. There might be several path conditions. Just give an SSA formula for one of them. (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 parameters passed to `foo`). Use, for each variable or parameter, the interval domain, i.e., the abstract elements depicted in the lattice to the right of `foo`. Observe this is not the sign domain discussed in the course. (2 pts)
3. Give, without justification, the weakest condition P such that executing the sequence of three assignments `z= x; x= y; y= z+z;` (where `x`, `y` and `z` are integer variables with no integer overflows) from a configuration satisfying P will always result in a condition satisfying the condition ($Q : x + y \geq 6$ and $x \leq y$). In other words, give the weakest (i.e., most general) predicate P such that the Hoare triple $\{P\}$`z= x; x= y; y= z + z;`$\{Q\}$ holds (i.e., is always true or valid). (2 pts)

Question 6: Security testing (6 points)

a) Consider a piece of small but highly safety-critical code (e.g., a component of the avionics software in an airplane). If you had to pick one of *concolic testing* and *greybox fuzzing* for automatic correctness testing of that software, which technique would you choose, and why?

b) When using a greybox fuzzer to test a program whose input format uses *checksums*, it is typically necessary to patch out (i.e., disable) the checksum validation code in the program. Briefly explain why.

c) State another fuzzing technique that would not require patching out checksum code prior to running tests, and briefly explain why it is capable of handling checksums.

d) Explain why just using a grammar to generate new test cases is typically not sufficient when using black-box generational fuzzing. What is needed in addition to the grammar?

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

The code on the next page shows a function `show_inventory` that is part of a system for viewing the present year's inventory history for a hypothetical military installation. The four types of tracked items in the inventory are: rifle ammunition, liters of diesel fuel, cans of beans, and liters of drinking water. The function asks the user to specify the day of year as an integer between 1 and 366 and the item of interest as a string, and displays the quantity of stockpiled units of the item the given day. The function also takes an argument that states the current user's clearance level, and checks that the user is authorized to view the inventory statistics for the given item. For example, drinking water requires only Restricted level, whereas diesel fuel requires Secret level clearance. (For simplicity, item names and associated clearances are hardcoded as constant arrays here.)

It can be assumed that the function is always called in such a way that the given clearance represents the actual clearance of the user. Moreover, it can be assumed that all comments in the code are truthful and correct.

The function has a serious security bug.

a) Identify the bug in the code, and state what kind of bug it is (out of the security bug types discussed in the course).

b) Clearly explain how the bug could be exploited, and what the consequences of a successful exploit would be (i.e., what the bug allows the attacker to do). State any (reasonable) assumptions you might make.

c) Show using code/pseudocode (and a clear motivation) how the bug should be fixed.

```c
typedef enum {
    CLR_RESTRICTED = 1,
    CLR_SECRET = 2,
    CLR_TOPSECRET = 3
} ClearanceLevel;

const size_t N_COLUMNS = 4;

// Names of columns in the inventory history database
const char* const COL_NAMES[] = {"ammo", "fuel", "beans", "water"};

// Corresponding security clearances required to view columns
const ClearanceLevel COL_CLEARANCE_LEVEL[] =
    {CLR_TOPSECRET, CLR_SECRET, CLR_RESTRICTED, CLR_RESTRICTED};

// Returns a malloc-allocated array of exactly 366 integers (one per day
// of year), representing each row value for the given column.
// (For non-leap years, the last element is always 0.)
// Returns NULL in case of error.
int* load_column(const char* col_name);

void show_inventory(ClearanceLevel clearance_level) {
    // Allocate array to hold pointers to columns
    int** columns = malloc(N_COLUMNS * sizeof(int*));

    // Fetch each column (as an array of 366 ints)
    for(size_t col = 0; col < N_COLUMNS; col++) {
        columns[col] = load_column(COL_NAMES[col]);
        // Column lookup should never fail. Kill program if it does.
        assert(columns[col] != NULL);
    }

    char item[32];
    int day;

    // Loop until explicitly terminated
    while(1) {
        item[0] = 0; // Empty string
        day = 0;

        printf("Input day of year (1-366, or 0 to quit):\n");
        // Parse user input as one integer and store that int into 'day'
        scanf("%d", &day);
        if(day > 366) {
            printf("Invalid day. Try again.\n");
            continue;
        } else if(day == 0) {
            printf("Exiting.\n");
            break;
        }
        printf("Input item name:\n");
        // Read string of characters from standard input into 'item' until first
        // whitespace. At most 31 characters are read and a NULL-terminator
        // is always appended.
        scanf("%31s", item);
        size_t i;
        for(i = 0; i < N_COLUMNS; i++) {
            // Check if given item name matches column name
            // (strcmp returns 0 on exact match)
            if(strcmp(item, COL_NAMES[i]) == 0) {
                if(clearance_level < COL_CLEARANCE_LEVEL[i]) {
                    printf("Insufficient clearance for item. Try again\n");
                } else {
                    printf("Quantity of stockpiled %s at day %d: %d\n",
                        item, day, columns[i][day-1]);
                }
                break;
            }
        }
        // Did we reach end of for-loop without any match?
        if(i == N_COLUMNS) {
            printf("Unkown item name. Try again.\n");
        }
        printf("\n");
    }

    for(size_t col = 0; col < N_COLUMNS; col++) {
        free(columns[col]);
    }
    free(columns);
}
```