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

Distance exam

TDDC90 Software Security

2022-01-15

Teacher on duty

Ulf Kargén, ulf.kargen@liu.se, 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 36. 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

Answers should be submitted through Lisam as a single PDF or ASCII text (.txt) document before the end of the exam time. If you don't have access to a good software for making technical drawings that you are already familiar with, it is recommended to draw figures by hand and scan/photograph them. Figures could either be integrated into the PDF, or submitted as separate files in JPEG or PNG format. If you chose the latter approach, file names should be of the format Figure1, Figure2, and so on. All attached figures should be clearly referred to in the text by their file name. To facilitate easier grading, it is preferred that you express formulas, program code, etc. as text, and not as handwritten figures.

You are allowed to use any aid, but **all kinds of collaboration with others is strictly forbidden**. Also, **copying any part of an answer from another source will be considered plagiarism**. The questions will be checked for plagiarism and sharing of answers between students using Urkund, and any suspected cheating will be reported to the university disciplinary board. **By taking the exam, you solemnly promise to abide by the above rules.**

In the event that you experience technical difficulties with Lisam that prevent you from submitting, it is allowable to submit answers via email. However, this should only be used as a last resort if Lisam for whatever reason would stop functioning.

The distance exam will not be anonymous due to the exceptional COVID-19 situation.

Ulf Kargén will be available to answer questions during the exam via email and phone.

Question 1: Secure software development (4 points)

a) *Attack surface reduction* is an important principle in secure software design. Give a practical example of this principle being applied in a software system. Provide enough details and explanation to make it clear how your example relates to attack surface reduction.

b) Consider a hypothetical software development project. In an effort to minimize the risk of successful buffer overflow exploits against the developed software, the team leader decides to both ban all use of `strcpy` and related library functions, and that all software components must also be compiled with stack cookies enabled. What other important principle of secure software design is this an example of? Motivate your answer.

Question 2: Exploits and mitigations (5 points)

a) Imagine that you found a stack-based buffer overflow in a web browser, which allows you to overwrite a function pointer on the stack. By calling the built-in JavaScript function `String.search` in the vulnerable browser with the argument *exploit*(*A*), the function pointer is first overwritten with an address *A* of your choosing, and then a call is made by dereferencing the function pointer. (Here, *exploit* (*x*) refers to some way of crafting an appropriate input based on *x*.) Explain, using high-level pseudocode, the sequence of steps required to exploit the above vulnerability for arbitrary code execution on a system that uses ASLR but not DEP.

b) Consider a Heartbleed-style vulnerability, which allows an attacker to read some data past the end of a heap buffer. This kind of vulnerability can be used to disclose sensitive information on the heap stored adjacent to the buffer. In a sentence or two, explain whether ASLR would be effective at mitigating this kind of attack.

Question 3: Design patterns (2 points)

The two similar design patterns *Privilege Separation* and *Defer to Kernel* are both special cases of the more general pattern Distrustful Decomposition. Briefly discuss in which cases it is more appropriate to use one or the other.

Question 4: Web security (6 points)

a) Modern websites typically use TLS to encrypt all requests and responses to/from the web server. If a TLS-enabled website also allows regular unencrypted HTTP requests, it is generally considered a security vulnerability. Give an example of how an attacker could exploit such a configuration error to gain access to a logged-in user's account at a website. Make sure to state all prerequisites for the attack, and any assumptions you make.

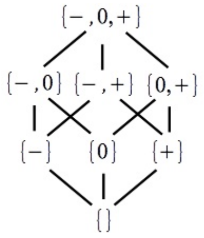
b) In the course, we mentioned a technique that can mitigate the consequences if a TLS-enabled website is mistakenly configured to also accept unencrypted HTTP requests. Name it and briefly explain how it works.

c) It has sometimes been incorrectly stated that disallowing GET-requests, and instead only allowing POST, is a way to mitigate cross-site request forgery (CSRF). Explain the rationale for this misconception. Also, give a counterexample of how CSRF-attacks can still be carried out.

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 x){
 2    int res = 1;
 3    int cur = 1;
 4    int old = 0;
 5    while(x > 0){
 6      old = cur;
 7      cur = res;
 8      res = cur + old;
 9      x = x - 1;
10      assert(x >= 0);
11    }
12    assert(res > old);
13    return res;
14  }
```



We aim to check the assertions `(x >= 0)` and `(res > old)` respectively at lines 10 and 12 in procedure `foo`. Questions:

1. Symbolic execution:

   (a) Give, without checking its satisfiability, the SSA formula for the path condition that corresponds to a sequence of instructions starting at line 1 of procedure `foo` and performing one iteration of the loop before violating the assertion at line 12. (1 pt)

   (b) Is this path condition satisfiable? Explain. (1 pt).

   (c) Is it possible to use symbolic execution to verify, in a finite amount of time, that the assertion at line 12 is never violated? Explain. (1 pt).

2. Abstract interpretation: Annotate, after convergence of the analysis, the end of each line in `foo` with the abstract element associated to each one of the defined variables. (1pt)

3. The following part has 3 questions. The three answers result in a minimum of 0 pt and a maximum of 3 pts. Each wrong answer counts negative. E.g., one correct answer (1 x 1 pt), one wrong (1 x -1 pt) and not answering one (1 x 0 pt) result in a score of 0 pt out of the 3 possible points. Giving two wrong answers (2 x -1 pt) and one correct (1 x 1 pt) gives 0 points.

   Here, `old=0` stands for assignment and `res == 1` for testing equality. The sequence `old=cur; cur=res; res=cur+old` represents sequential composition, i.e., the three assignments are performed in sequence starting with `old=cur`. State, without justification, whether each of the following Hoare-triples is true or false.

   (a) $\{res == 1\}$ `old=0` $\{res == 1\}$

   (b) $\{res \geq cur > old > 0\}$ `res=cur+old` $\{res > cur > old > 0\}$

   (c) $\{x > 0\}$ `while(x > 0){y=y-1; x=x-1;}` $\{x > 0\}$

Question 6: Security testing (6 points)

a) When using in-memory fuzzing, it is very important that the fuzzed interface does not alter global state (i.e., global variables whose values affect program behaviour). Explain why. Make sure to sufficiently explain how in-memory fuzzing works to clearly motivate your answer.

b) Greybox fuzzing is sometimes also called *evolutionary fuzzing*. Explain why.

c) Consider cross-site scripting (XSS) vulnerabilities. Which of the two vulnerability types Stored XSS and Reflected XSS is generally easier to detect using a black-box web application fuzzer? Clearly motivate your answer. (Make sure to include an explanation of the difference between Stored and Reflected XSS in your motivation.

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).

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");
        // Read one integer from standard input 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);
}
```