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

Written exam

TDDC90 Software Security

2024-01-13

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)

Explain by example how attack trees are created. Come up with a scenario on your own, and make sure that you explain all the details of attack trees that was covered in the course, including different types of nodes/edges and how annotations work.

Question 2: Exploits and mitigations (6 points)

For a given type of attack, a specific mitigation might be effective, meaning that it at least raises the challenge for an attacker in some way, or it might be completely ineffective, meaning that it is of no concern to the attacker. For each of the three mitigations below, clearly explain whether the mitigation is effective or not against exploitation attempts of *use-after-free bugs* for *arbitrary code execution*.

a) DEP

b) ASLR

c) Stack Cookies

Question 3: Design patterns (3 points)

Briefly explain how the *secure chain of responsibility* pattern works. What is the motivation and purpose of using it? When is it suitable to use it?

Question 4: Web security (7 points)

a) Using pseudo-code, write server-side code that contains a vulnerability that allows for an *SQL injection* attack. Your code should be detailed enough that it is clear how attacks can be made. Explain your code in English (or Swedish). Also give an example (including concrete attack input) of how an attacker could use the bug to attack a user of the affected site. Finally, explain how the code should be altered in order to mitigate the attack.

b) State a web attack discussed in the course that allows *arbitrary* code execution …

   i. … on the *web server*.

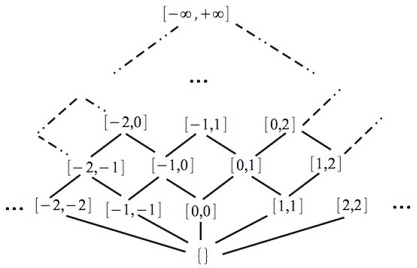
   ii. … in the *client's browser*.

   For each attack, briefly explain how it works, using a few sentences.

Question 5: Static analysis (7 points)

Assume `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows). Suppose that given an integer `x`, the expression (`x % 2`) returns 0 if `x` is even and 1 otherwise, i.e., (`x % 2`) returns 0 if there exists an integer k with a positive, negative or zero value such that $x = 2 \times k$. It returns 1 otherwise. Consider the following procedure.

```
1   int foo(int in){
2     int out = 0;
3     if(in < 0){
4       out = in - 1;
5     }else{
6       out = in + 1;
7     }
8     if((in % 2) != 0){
9       out = out + 1;
10    }
11    assert(in * out >= 0);
12    assert(!(-2 < out && out < 2));
13    return out;
14  }
```



We aim to check the assertions at lines 11 and 12 in procedure `foo`. Questions:

1. Symbolic execution. Give, without discussing their possible satisfiability, the SSA formula for each path condition through the procedure `foo` that satisfies the assertion at line 11 but violates the one at line 12. There might be several path conditions. Give an SSA formula for each one of them. (4 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. Use, for each variable, the interval domain, i.e., the abstract elements depicted in the lattice to the right of `foo`. Observe this is not the sign domain. (2pt)

3. Give, without justification, the weakest precondition P of the post-condition ($Q : out = 4 \times k$ for some integer $k > 0$) and the procedure `foo`. Observe the post-condition Q states *out* is a positive multiple of 4. In other words, give the weakest (i.e., most general) predicate P such that the Hoare triple $\{P\}$`foo`$\{Q\}$ holds (i.e., is valid). (1pt).

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) Consider the two fuzzing techniques *black-box mutational fuzzing* and *greybox fuzzing*.

   i. Explain what the main similarity between the two is.

   ii. Explain the principal difference between the two, which is the reason for the generally much better code coverage of greybox fuzzing.

   iii. Briefly describe one case (i.e., a specific kind of code construct) where both techniques are likely to yield poor code coverage.

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

The code on the next page shows part of a simple API for working with buffers containing Unicode text. The code has a serious bug, which could potentially be exploited by attackers.

a) Identify the vulnerability and explain how an attacker could trigger the bug. You *don't* need to explain how to craft an exploit for the bug.

b) Explain, using English/Swedish *and* code/pseudocode, how to fix the bug.

```
/*** External library dependencies ***/

/* Checks if pointed-to data is a valid Unicode character.
   Returns 1 if a valid character, 0 otherwise.
   Details of how this function works is unimportant. */
int isValidUnicode(const char* data);



/*** UnicodeBuf API definitions ***/

#define BUF_SIZE 1000000

/* Data structure to represent a buffer with Unicode data */
struct UnicodeBuf {
  char* buffer;         // Buffer to hold text data
  unsigned int n_used;  // Number of characters in buffer
};

/* Allocates and returns a new empty UnicodeBuf */
struct UnicodeBuf* UBCreate() {
  struct UnicodeBuf* b = malloc(sizeof(struct UnicodeBuf));
  b->buffer = malloc(BUF_SIZE);
  b->n_used = 0;
  return b;
}

/* Destroys an existing UnicodeBuf */
void UBDestroy(struct UnicodeBuf* b) {
  free(b->buffer);
  free(b);
}

/* Appends Unicode string (2 bytes per character) to UnicodeBuf.
   'data' can always be assumed to hold 2 * 'n_chars' bytes.
   Checks that each copied character is valid Unicode.
   Returns 0 on success, or non-zero to indicate an error. */
int UBAppend(struct UnicodeBuf* b, char* data, unsigned int n_chars) {

  if(b->n_used * 2 > BUF_SIZE - (n_chars * 2))
    return 1; // Too much data

  unsigned int i;
  for(i = 0; i < n_chars; i++) {
    if(isValidUnicode(data + 2 * i)) {
      b->buffer[2 * b->n_used]     = data[2 * i];
      b->buffer[2 * b->n_used + 1] = data[2 * i + 1];
    } else
      return 2;

    b->n_used += 1;
   }

  return 0;
}
```