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

Written exam

TDDC90 Software Security

2023-01-14

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)

Consider the following two ways a developer might attempt to increase the security of a piece of software:

a) Using the Privilege Separation design pattern.

b) Compiling the software with Control-Flow Integrity enabled.

For each of the secure-development principles *attack-surface reduction* and *defense in depth*, explain which of the two strategies a) and b) above that most closely align with the principle. Clearly motivate your answer. (A simple mapping with no motivation gives no points.)

Question 2: Exploits and mitigations (5 points)

a) ROP could be considered a generalization of the older attack technique *return-to-libc*. Explain why.

b) Clearly explain why DEP and ASLR must always be used together in order to be effective.

Question 3: Design patterns (4 points)

a) What is the motivation and purpose of the *secure chain of responsibility* pattern? When is it suitable to use it?

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

Question 4: Web security (6 points)

a) Consider the Client Side Request Forgery (CSRF) and Server Side Request Forgery (SSRF) attacks. What do they have in common, and how do they differ?

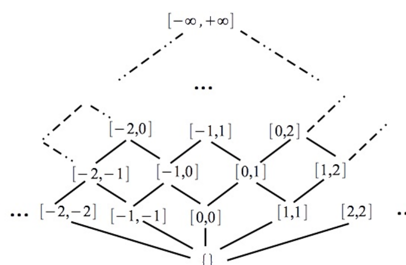
b) Explain, using an example, how an XXE vulnerability could be exploited to perform an SSRF attack. Your example should be detailed enough, so that it is possible to understand every principal step of the attack.

c) Briefly explain why using *salts* is important when storing passwords for a web app.

Question 5: Static analysis (7 points)

Assume `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows). Recall `%` is the modulo operator (`x % 2` returns 0 if `x` is even and 1 otherwise). Consider the following procedure.

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



We aim to check the assertions at line 11 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`, getting in the "then" branches of both if statements, satisfying the first assertion (line 11) and violating the second assertion (line 12). (1 pt)

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

   (c) How many paths should symbolic execution check in order to verify the assertion at line 12 always holds? Explain. (1 pt).

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`. (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, all variables are integers; `out = 0` stands for assignment and `(out == 2 * in)` stands for the predicate that evaluates to true iff `out` is twice `in`. The sequence `out = 0; i = in; while(0 < i){out = out + 2; i = i - 1;}` represents sequential composition of two assignments followed by a while statement. State, without justification, whether each of the following Hoare-triples is valid or not.

   (a) $\{0 \leq in\}$ `out = 0; i = in; while(0 < i){out = out + 2; i = i - 1;}` $\{out == 2 * in\}$

   (b) $\{true\}$ `out = 0; i = in; while(0 < i){out = out + 2; i = i - 1; }` $\{out == 2 * in\}$

   (c) $\{true\}$ `out = 0; i = in; while(0 > i){out = out + 2; i = i - 1;` $\{out == 0\}$

Question 6: Security testing (6 points)

a) Briefly explain two reasons why automated fuzzing of web applications is often harder than fuzzing e.g. a desktop program written in C.

b) Explain what *sanitizers* are in the context of software security testing, and why they are used.

c) Give an example (using pseudocode and English/Swedish) of a program with a bug that would be easy to trigger with a *generation-based* fuzzer, but hard or impossible to trigger with any other fuzzer type discussed in the course (i.e., black-box mutational, greybox, and concolic testing/whitebox fuzzing). Clearly motivate your answer, including an explanation of why each of the other types of fuzzers would fail to find the bug.

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

The code on the next page shows part of a simple game engine implemented in C++. There are two types of units in the game: tanks and soldiers. A solder can either be a standalone unit, or be assigned as the driver of a tank. The code includes classes for representing units, and a GameWorld class, which keeps track of active units and has an interface for handling an attack at a given coordinate. The GameWorld class is supposed to be called by the main game loop, which is not shown here. There is also a function that handles joining of new players. Players are allowed to join while the game is already running.

To solve this problem, you don't need any knowledge of library functions or specific language features of C++, other than what is given in the comments. You can assume that all statements made in comments are correct.

The code contains at least one serious vulnerability, which could be exploited for arbitrary code execution.

a) Identify the vulnerability and give the name of this type of security bug.

b) Give a high-level description of what an arbitrary code execution exploit for this bug would look like. (That is, explain what is needed to trigger the bug, and conceptually explain how to manipulate the program for execution of shellcode.) For simplicity, you can assume that DEP and ASLR are not used. Clearly state all other assumptions you make.

c) Explain how to fix the bug.

```
// Abstract base class for units (tanks and soldiers)
class Unit {
protected:
  char* name;
  int pos_x, pos_y;

public:
  // Constructor
  Unit(char* unit_name, int x, int y) { name = unit_name; pos_x = x; pos_y = y; }

  // Polymorphic/virtual functions that should be implemented by derived classes
  virtual void move(int x, int y) = 0;
  virtual void attack(int x, int y) = 0;

  // Virtual function to check if unit is at given position (x, y)
  virtual bool atPos(int x, int y) { return pos_x == x && pos_y == y; }

  // Destructor. Called automatically when doing 'delete' on object,
  // right before deallocating the object.
  virtual ~Unit() {
    // Deallocate name. (With free, because allocated with strdup. Does nothing if name==NULL)
    free(name);
  }
};

class Soldier : public Unit {
public:
  Soldier(char* unit_name, int x, int y) :
    Unit(unit_name, x, y) /* Call base-class constructor */ {}
  virtual void move(int x, int y) { /* Details unimportant */ }
  virtual void attack(int x, int y)  { /* Details unimportant */ }
};

class Tank : public Unit {
protected:
  Soldier* driver;
public:
  Tank(char* unit_name, Soldier* tank_driver, int x, int y) :
    Unit(unit_name, x, y) { driver = tank_driver;  }
  virtual void move(int x, int y) { /* Details unimportant */ }
  virtual void attack(int x, int y)  { /* Details unimportant */ }

  // Destructor for Tank class. Base-class destructor is called
  // automatically after this one, but before object is deallocated.
  virtual ~Tank() { delete driver; /* Deallocate driver. */ };
};

class GameWorld {
protected:
  set<Unit*> units; // C++ standard set for holding pointers to active units

public:
  void add_unit(Unit* u) { units.insert(u); /* Insert pointer to unit into units set */ }

  // Handle attack at coordinates (x, y)
  void handle_attack(int x, int y) {
    // Iterate over all active units. (If you are unfamilier with C++ iterators,
    // you can just assume that handling of iterators here is correct.)
    for(auto iterator = units.begin(); iterator != units.end(); ) {
      Unit* unit = *iterator; // Get unit pointer from iterator
      // Check if unit was destroyed
      if(unit->atPos(x, y)) {
        // Remove unit pointer from units set and move iterator to next element
        units.erase(iterator++);
        delete unit; // Call destructor(s) and deallocate unit
      } else {
        ++iterator; // Just move iterator to next element
      }
    }
  }
};

// Called when a new player joins. (Players can join while game is already running.)
void new_player(GameWorld* world, const char* alias, int x, int y, bool with_tank) {
  // The C library function strdup will allocate new memory of accurate size (length + 1),
  // and copy given string into it. A pointer to the new memory is returned.
  char* name = strdup(alias);
  if(with_tank) {
    Soldier* driver = new Soldier(name, INT_MAX, INT_MAX); // Use dummy position
    world->add_unit(driver);
    Tank* tank = new Tank(NULL, driver, x, y);
    world->add_unit(tank);
  } else {
    Soldier* solider = new Soldier(name, x, y);
    world->add_unit(solider);
  }
}
```