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

Written exam

TDDC90 Software Security

2019-01-15

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) For *each* of the two modelling techniques *misuse cases* and *attack trees*, state in which part of the software development lifecycle it is most appropriate to use the technique. Briefly motivate your answers.

b) Name Phase 3 of Microsoft's SDL (not counting the pre-SDL phase) and briefly explain the activities prescribed by SDL in this phase.

Question 2: Exploits and mitigations (5 points)

a) 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.

b) Briefly compare the two attack techniques *return-to-libc* and *ROP*. How are they related? What is the main difference between them?

c) 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)

a) Explain the design pattern *privilege separation*. Your answer should include a diagram, pseudo-code and an explanation of why and when the pattern should be used.

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

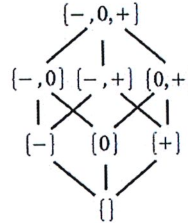
b) Explain, in a sentence or two, what a *command injection* vulnerability is.

c) Using pseudo-code, write server-side code that contains a vulnerability that allows for XSS. Your code should be detailed enough that it is clear how XSS attacks can be made. Explain your code in English (or Swedish). 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.

Question 5: Static analysis (7 points)

The following function computes the nth factorial number, i.e., product(n) $= 1 \times 2 \times 3 \ldots \times n$. Here, `int` denotes integers with an absolute value that can be arbitrarily large (i.e., no integer overflows).

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



We aim to check the assertions (p >= 2^{i-1}) at line 9 and (0 <= i) at line 11.
Questions:

1. Consider first the assertion (0 <= i) at line 11:

   (a) Symbolic execution: 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 11 and violating the assertion there (i.e. violating the (0 <= i) assertion). (2 pt)

   (b) Abstract interpretation: can an abstract interpretation analysis based on the sign abstract domain mentioned above establish that the assertion is never violated? explain by annotating each line with the abstract element associated to each variable (i.e., each one of i, p and n) and obtained at the end of such an analysis (i.e., after the analysis converges). (1pt)

2. Consider now the assertion (p >= 2^{i-1}) at line 9:

   (a) What does it mean for the predicate $wp(stmt, Q)$ to be the weakest precondition of a predicate Q with respect to a program statement $stmt$? (1 pt)

   (b) Give P_8 defined as the weakest precondition of the predicate (p>= 2^{i-1}) with respect to the assignment `i = i + 1` at line 8; then give P_7 defined as the weakest precondition of the predicate P_8 with respect to the assignment `p = p * i` at line 7. (2pt)

   (c) Is P_7 an invariant of the loop? would having P_7 as an invariant of the loop be enough to establish the assertion at line 9? justify. (1pt)

Question 6: Security testing (6 points)

a) Greybox fuzzing is often exponentially faster at finding bugs than a simple mutational black-box fuzzer. Explain why, using a small example.

b) In a sentence or two, explain what *magic constants* are in the context of fuzzing, and why they pose a problem to fuzzers that use random input generation.

c) What is the benefit of using a dynamic-analysis tool such as Valgrind or AddressSanitizer during fuzzing? What is the downside?

Question 7: Vulnerabilities in C/C++ programs (7 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 is 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);
  }
}
```