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

Written exam

TDDC90 Software Security

2023-03-17

Solution hints

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

Important: Note that the purpose of the solution hints is to provide guidance when using the old exam for self-study. For full score on an exam, more detailed/elaborate answers than the one-sentence hints given here would typically be required.

Question 1: Secure software development (4 points)

For each of the two modelling techniques *misuse cases* and *attack trees*, briefly (using a few sentences each) explain what the technique is used for. Also state in which part of the software development lifecycle it is most appropriate to use the technique, and briefly explain why.

Misuse cases: Model ways in which a system can be misused/abused. Can be used to elicit security requirements (SDL Phase 1).

Attack trees: Model different ways in which a particular attack against a system can be performed. Can be used for risk analysis and threat modelling (SDL Phase 2).

Question 2: Exploits and mitigations (5 points)

a) Could ROP be used to exploit a buffer overflow on the *heap*? If no, explain why it is impossible. If yes, explain how.
   Yes, using *stack pivoting*… (Would need explaining for full score.)

b) Consider a Heartbleed-style vulnerability, which allows an attacker to read some data past the end of a buffer. This kind of vulnerability can be used to disclose sensitive information stored adjacent to the buffer. For each of the following two mitigations, explain whether or not it can mitigate this kind of attack, and why.

   i. ASLR
      No, only randomizes offset between memory segments, not content of heap

   ii. DEP
      No, only prevents execution of injected shellcode, not reading out of bounds

Question 3: Design patterns (4 points)

a) Explain the design pattern *secure factory*, including an explanation of why and when the pattern should be used.
   Create object based on supplied security credentials. Intent is to separate the security dependent logic for creating an object from the functionality of the created object. Applicable when system creates different versions of objects depending in security credentials.

b) Both of the patterns *privilege separation* and *defer to kernel* are specialized versions of a more general design pattern. Give the name of this general pattern, and explain the intent and motivation of it in 2-3 sentences.
   Distrustful Decomposition. Intent is to separate functionality of a system into mutually untrusting components. Motivation is to reduce risk of exploits affecting entire system, if a security bug exists within some part of the system.

Question 4: Web security (6 points)

a) Consider a web app using Java as the server-side language. The web app needs to implement serialization of a complex data type (for example, a Java class), so that objects of the type can be received in web requests (for example, as part of an API). Clearly explain why it is inappropriate from a security point of view to implement this functionality with Java's own built-in serialization. Also, briefly explain how the functionality could be implemented in a secure way.
Java's serialization feature assumes that serialized data comes from a trusted source. Using it in a web app would lead to an *insecure deserialization* vulnerability, since an attacker could send requests with malicious serialized code, or malformed serialized objects that trigger e.g. memory corruption in deserialization component. Instead, implement own serialization using format with only primitive data types (no serialized code). For example, JSON could be used.

b) 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.
Easier to do CSRF with GET, since it is enough to send malicious link in e.g. email. Still possible with POST, e.g., by sending link to attacker-controlled page, which in turn redirects to target web app with a malicious request.

Question 6: Security testing (6 points)

a) Which of *cross-site request forgery* (CSRF) and *SQL-injection* would be easier to detect (in the general case) using an automated web application fuzzer? Clearly explain your reasoning.
SQL injection would be easier. Can try requests containing SQL and look for SQL output in reply. CSRF would require human-level understanding of which parts of web app that should not be accessible via guessable URLs.

b) What is the path explosion problem in concolic testing?
Number of paths increases exponentially with number of branches (≈ code size). Concolic executor likely to get stuck exploring small part of a program, if systematically exploring each path.

c) A generation-based fuzzer generally requires two components to work: a *grammar* and a set of *fuzzing heuristics*. Explain the purpose of both of these components.
Grammar defines structure/syntax of input format. Used to generate new inputs. Fuzzing heuristics are certain "extreme" input fragments, known to often trigger bugs. Applied while generating inputs from grammar to create semi-valid inputs.

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

The small C++ function shown on the next page reads text from a file, one line at a time, and concatenates all lines into one string before proceeding to process the concatenated text. (The nature of that processing is not relevant here.) The code contains at least one serious vulnerability, which could potentially be exploited to allow arbitrary code execution.

a) Identify the vulnerability and state the vulnerability type.
Integer overflow in `if(total_read + len > MAX_DATA)`. (The expression can never be false, since MAX_DATA = INT_MAX, and variables are of type int.) Leads to heap-based buffer overflow in `strcat` call.

b) Explain what the input file should look like in order to trigger the bug. (You *don't* need to explain in detail how to exploit the vulnerability.)
A large ASCII file with each line < `BUFSIZE` characters, but where the sum of all strings' lengths is > `INT_MAX`

c) Explain, in words *and* code/pseudocode, how to fix the bug.
Change type of all constants and variables representing lengths to `size_t`. Add a check for integer overflow prior the check against `MAX_DATA`.

You can assume that the comments in the code correctly describe the behaviour of library functions, etc. You don't need any additional knowledge about the library functions used than what is given in the comments.

```
void read_and_process(istream& in_file)
{
   const int MAX_DATA = INT_MAX;
   const int BUFSIZE = 1000;

   // Allocate a buffer on the heap with MAX_DATA bytes.
   char* concatenated = new char[MAX_DATA];
   // 'concatenated' should initially contain an empty string.
   concatenated[0] = 0;
   int total_read = 0;

   char buffer[BUFSIZE];

   // 'getline' reads one line of text (until a line delimiter is reached)
   // from 'in_file' into 'buffer'.
   // A maximum of 'BUFSIZE' bytes is written to 'buffer', including the
   // null terminator.
   // If end-of-file is reached, or the current line contains 'BUFSIZE'
   // characters or more, the call to 'getline' will evaluate to false,
   // and the loop will be terminated.
   while(in_file.getline(buffer, BUFSIZE))
   {
      int len = strlen(buffer) + 1;
      if(total_read + len > MAX_DATA)
      {
         printf("Error: Too much data");
         exit(1); // Quit program
      }

      // Append string in 'buffer' to existing string in 'concatenated',
      // starting from the position of the old null terminator in
      // 'concatenated'.
      strcat(concatenated, buffer);
      total_read = strlen(concatenated);
   }
   // Complete string read. Process it...
   process(concatenated);
}
```