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

Written exam

TDDC90 Software Security

2024-03-15

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)

a) Using a few sentences, describe the principle of *defense in depth*.

b) Map each of the following two activities to the most appropriate phase of SDL, and explain why it is most appropriate to perform the activity during that phase: *fuzz testing* and *static analysis*.

Question 2: Exploits and mitigations (6 points)

a) Most implementations of stack cookies also reorder the position of local variables on the stack in a specific way. Explain in what way variables are reordered, and the motivation for doing this.

b) Name one attack technique that is prevented by CFI but not by DEP. Explain why CFI is effective for stopping the attack and why DEP isn't.

c) In the context of software exploits, explain what *stack pivoting* is, and why it is used.

Question 3: Design patterns (3 points)

Come up with a small example that demonstrates use of the *secure factory* design pattern. Your answer should include code/pseudocode and an explanation in English/Swedish.

Question 4: Web security (6 points)

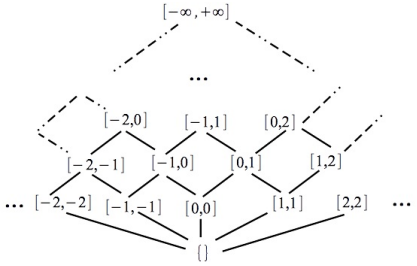
a) Using pseudo-code, write server-side code that contains a vulnerability that allows for *stored* XSS. Your code should be detailed enough that it is clear how XSS attacks can be made. Explain your code in English (or Swedish). Give a (realistic) example of how an attacker could use the bug to attack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

b) 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.

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 a, int b){
2    int rslt = 0;
3    if(a >= b){
4      rslt = a - b;
5    }else{
6      rslt = b - a;
7    }
8    assert(rslt >= 0);
9    return out;
10 }
```



Questions:

1. Symbolic execution. Give, without discussing their possible satisfiability, the SSA formula for each path condition through the procedure `foo` that violates the assertion at line 8. Note that there might be several path conditions. Give an SSA formula for each one of them. (2 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 (including the parameters passed to `foo`). Use, for each variable or parameter, the interval domain, i.e., the abstract elements depicted in the lattice to the right of `foo`. Observe this is not the sign domain. (2 pts)

3. Give, without justification, the weakest precondition P of the post-condition ($Q : x+y \geq 10$ and $x-y \geq 0$ for the sequence `z= x; x= y; y= z;` where `x`, `y` and `z` are integer variables like in `foo` (i.e., no integer overflows). Observe this question is about the sequence `z= x; x= y; y= z;`, not the procedure `foo` mentioned in the previous two questions. In other words, give the weakest (i.e., most general) predicate P such that the Hoare triple $\{P\}$`z= x; x= y; y= z;`$\{Q\}$ holds (i.e., is always true or valid). (3 pts)

Question 6: Security testing (7 points)

a) Imagine that you are tasked with creating a *black box* fuzzer for a JavaScript interpreter. Which fuzzing strategy (mutation or generation) would be most appropriate? Clearly motivate your answer.

b) Consider the problem of fuzzing a fixed-function hardware device, e.g., a chip soldered onto a circuit board in some appliance (assuming that you have a way to send data to and receive output from the device). For each of the three following fuzzing/testing techniques, state and briefly motivate whether it would be a viable choice in this case:

   i. Greybox fuzzing

   ii. Concolic testing

   iii. Black-box mutational fuzzing

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

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

The next page shows code for a simple utility `add_user`, which works similar to the Unix `useradd` command, but which is intended for situations where also non-root users must be able to add new accounts to the system. Since the tool must be able to change the password file, among other things, it is configured as SUID root, meaning that it can perform operations requiring root privileges even if it is invoked by a regular user. The tool takes the name of the new account as a mandatory argument. By default, the tool creates a regular user account. An optional argument "`--root`" can also be supplied for creating a new account with root privileges. This option, however, is restricted to work only if a root user is invoking the tool, to prevent privilege escalation.

The program has an implementation error, which constitutes a serious security vulnerability.

a) Identify the bug in the code and name the vulnerability type.

b) Explain how an attacker could exploit the bug. Your answer should show in detail what the input to the program must look like in order to succeed with the exploit, and what the exploit leads to. State any assumptions about, e.g., memory layout that you make.

c) Explain how to fix the bug.

```c
/* Add regular user account with provided username.
   Queries for password for new account when called.
   Details unimportant. */
void add_account(const char* username);

/* Add root-privilege account with provided username.
   Queries for password for new account when called.
   Details unimportant. */
void add_root_account(const char* username);

int main(int argc, char** argv)
{
    size_t idx;
    int restrict_root = 1;
    char username[32];

    // Remove restrictions if root user is invoking program, indicated by
    // a real user ID of 0. (The effective user ID is always 0 since the
    // the program is SUID root.)
    if(getuid() /* get real UID */ == 0)
        restrict_root = 0;

    // Check argument count.
    // First argument (argv[0]) is always name of executable,
    // second argument (argv[1]) should be username
    if(argc < 2) {
        printf("Too few arguments. Usage: add_user USERNAME [--root]\n");
        exit(1); // Exit with error code
    }

    if(strlen(argv[1]) > 32) {
        printf("Usernames must be at most 32 letters\n");
        exit(1);
    }

    strcpy(username, argv[1]);

    // Username must only contain lowercase characters,
    // according to our system policy
    idx = 0;
    while(username[idx] != 0) {
        username[idx] = tolower(username[idx]);
        idx++;
    }

    // Check for third optional "--root" argument
    if(argc > 2 &&
       strcmp("--root", argv[2]) == 0
       /* strcmp returns 0 if strings are equal */)
    {
        if(restrict_root) {
            printf("Access is restricted! Are you logged in as root?\n");
            exit(1);
        }
        add_root_account(username);
    } else {
        add_account(username);
    }

    return 0;
}
```