Computer Security (COM-301)

Software Security

MCQ Final 2022

Which of the following statements are true:

a) The WˆX policy, when strictly enforced, prevents code injection attacks.

b) Canaries aim to protect from both code injection and control flow hijacking attacks.

c) Control flow hijacking attacks are possible even when the adversary does not know the addresses of the system functions.

d) A fuzzing testing framework must choose between attempting to test all control flows or test all data flows.

A, B, and C are true.

D is wrong since it is *possible* to try to check both, even if not common in practice.

Side Channels

```
1: int calculate(char* input) {
2: char secretkey[128] = [0, 1, 0, 1, 1, ...];
3: int output = 0;
4: for(int i = 0; i < 128; i++) {
5:     if (secretkey[i] == 0) {
6:         output += fast_function(input);
7:     }
8:     else if(secretkey[i] == 1) {
9:         output += slow_function(input);
10:     }
11: }
12: return output;}
```

Describe a **testing methodology** based on two techniques seen in the class: fuzzing and sanitization to test whether `calculate`() is vulnerable to timing attacks or not. If the testing technique cannot be used to assess the vulnerability, explain why.

First, timing attacks: Observing the execution time, therefore, leaks whether the current bit of the secret key is zero or one.

How to test timing attacks:
A simple fuzzer would technically work here, as long as we know where the position of the 0 and 1 bits are. However, we need a second test to see whether the time difference is actually correlated with 0s and 1s. I.e. the code may not be running in constant time for different bits, but it can be running at random durations that are inconsistent with whether a key bit is 0 or 1. As a result, we can use a simple fuzzer to generate the inputs but we either have to observe the second part manually or have a more complicated script that runs the consistency check.
Note: While doing fuzzing we only have control over the inputs of the program. Since key is not an input here, it is not possible to try different keys using fuzzing techniques.

Sanitization does not help: it would not be able to measure the running time of the static data that we feed into. This is not the role of a sanitizer. Sanitizers detect memory corruptions, race conditions, but are not measurement tools. In fact, due to the overhead they introduce, would hinder any time measurement on the program.

Mission failed.. successfully !

Gru has written the code below to manage the bonuses of minions that participate in evil missions. Gru asks for your help debugging the function. Identify two lines that contain unsafe code that may lead to a memory safety error. For those two lines i) explain the vulnerability, and ii) explain whether a Minion can exploit this vulnerability to increase their bonus even when their mission did not succeed.
Assume that Minions are good teammates and will never steal a bonus from another Minion by providing others' successful mission names.

```
int bonus[100] = { 0 }; /* array of 100 integers initialized to 0 */
char successDB(char* mission) { /* function that returns 0 for failed
missions and 1 for successful missions. If the mission does not exist, it
crashes */ }

1: int AddMission(int minionID) {
2:            char success;
3:            char mission[30];
4:            gets(mission); /* minion provides name of mission */
5:            success = successDB(mission);
6:            if (success == 1) { bonus[minionID] += 1 };
7:            return 0; }
```

L4: Vulnerability that leads to memory safety error: Minion can overflow the mission name with their keyboard input. Would it lead to increase in bonus: It depends (on some assumptions and the exact argument.
Possible argument: Yes it can lead to increase in bonus. The call to `gets` provides a way for the user to write on the stack (if the string as input gets more than 30 chars).

L6: Vulnerability: minionID is out of bound (100). Increase in bonus: It depends.