Grenoble INP **UGA**
Master CybserSecurity & Master MoSiG **Year 2016-17**

Software Security & Secure Programming

Written Assignment - Tuesday November the 8th, 2016

Duration : 60 minutes – **Authorized documents :** one A4 sheet of paper

Exercise 1. (∼ 8 pts)

The C language authorizes explicit and implicit conversions (i.e., with or without a cast) between integers. For instance `short int` may be converted into `long int` (and conversely), and `signed int` may be converted into `unsigned int` (and conversely). However, according to the CERT secured coding standards, such conversions must be guaranteed *not to result in lost or misinterpreted data*.

Let us consider for instance the following function `func` which takes as a parameter `param` an `unsigned long int` value and converts it as a `signed char` before using it :

```
void func(unsigned long int param) {
  signed char sc;
  sc = (signed char) param;  /* explicit type conversion */
  /* ... */   // use now sc instead of param
}
```

Q1. Assuming that `long int` are encoded on 32 bits and `char` are encoded on 8 bits explain why this function is insecure (giving an example of user-provided value for `param` producing *lost or misinterpreted data*).

Q2. Give an example of *vulnerability* that may occur within function `func` due to the problem raised in question 1. You don't need to give a complete code example (nor to fully respect the C syntax), but you should clearly indicate how this vulnerability is triggered, and what is the potential gain for an attacker.

Q3. How to modify the function `func` in order to keep the type conversion but to warn the user in case of insecure behavior ? (give the new version of this function).

Q4. According to the CERT, *the only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness.*

1. explain why this assertion holds ;
2. do you think such a property could be verified at compile-time ?

(to be continued on next page)

Exercise 2. ($\sim$ 6 pts)

One of the protection mechanisms that could be offered by the underlying execution platform of a program is to check that the *control flow integrity* (CFI) is preserved at runtime. This consists in verifying that the execution sequence actually followed on the binary code does correspond to a path of the program control-flow graph. In particular this allows to detect when the following property φ is violated :

$\varphi \equiv$ "when a function call terminates, the execution resumes where this function has been called"

Q1. Give a (short) example of a vulnerable program that could be mitigated with CFI checking.

Q2. When CFI checking is not provided by the execution platform, property φ can still be enforced by strengthening the initial code (i.e., adding extra-code to detect property violation). Explain how this can be done and illustrate it on your example.

Q3. Does CFI checking protect against *use-after-free*? Why (or why not)?

Exercise 3. ($\sim$ 6 pts)

Let us consider the following program, where the `copy` function is a (naive) attempt to protect the execution against buffer overflow vulnerabilities :

```
void copy(char b[], int l){
// b is a string and l is its length
    char t[16] ; // 16 bytes
    int ok ;    // 4 bytes

    if (l > 15)
        ok = 0 ;
    else
        ok = 1 ;
    strcpy(t, b) ; // copy b into t
    if (ok==0) {  // a buffer overflow did occur in t
       printf("a buffer overflow occured !");
       exit(0);
    } else  // t contains no more than 15 characters (no overflow)
       foo(t);
}

int main() {
   char buf[24];
   scanf("%s", buf) ;    // read a string value from the user into buf
   copy(buf, strlen(buf)) ; // strlen(buf) is the number of characters in buf
   /* ... */
   return 0;
}
```

Q1. Draw the execution stack when function `copy` is executed (assuming that local variable are put in the stack in the order they are declared).

Q2. This program is not secure : there exists a user input allowing to call `foo` with an array argument `t` containing more than 15 characters. Give an example of such input.

Q3. Give two examples of "protections" (either code-level, compiler-level or execution platform-level) that may detect/prevent this kind of code weakness (i.e., unsafe buffer copy function).