Grenoble INP **UGA**
Master CybserSecurity & Master MoSiG **Year 2018-19**

Software Security & Secure Programming

Written Assignment - Wednesday November the 14th, 2017

Duration : 60 minutes – **Authorized documents :** one A4 sheet of paper – Figures on next page

Exercise 1. (∼ 8 pts)

We consider the C code given on Figure 1, where `process()` is an external function we do not care about. Here the programmer tried to prevent possible attacks by carefully checking the parameters of function `copy_and_process()`. However, (at least !) two possible problems may still occur at runtime ...

Q1. Indicate two (or three ?) possible vulnerabilities in the code of in Figure 1.
Clue : in C, overflow between **signed** integers is an **undefined behavior**, which can be **freely** interpreted by compilers/optimizers ....

Q2. For each vulnerability you found, explain :
— how to trigger them, i.e., **with which specific input** and/or **under which specific conditions**) ;
— which gain an attacker could get if he/she manages to trigger the vulnerability.

Q3. Update this code to make it secure (while preserving the same "nominal behavior").

Q4. Is there some external "protections" allowing to make the code given on Figure 1 "secure", without modifying it ? Explain your answer in a few lines ...

Exercise 2. (∼ 8 pts).

We consider a Java Class `C1` with a public method `m1()` allowing to perform some computations on a **secret** resource `key` and returning some integer value. Clearly, this method should **not** be called by any **untrusted** caller. To ensure that, the caller should provide as a parameter to `m1()` some credential as a string `s`. A check is performed within `m1()` to verify that the caller is legitimate. When it is the case, permission `P`, allowing to read `key` is granted. Later on this permission is disabled (when no longer required). The corresponding code (in pseudo Java) is given on Figure 2.

Q1. Why is it necessary/useful to explicitly enable permissions to read `key` inside `m1()` (since the caller credential is already explicitly checked beforehand) ? Indicate in which conditions enabling this permission is required or not required ...

Q2. The way permission `P` is enabled/disabled inside `m1()` is clearly **insecure**. Indicate why, and how to correct it.

Q3. If this code was written in C or C++, it would **not** be possible to enable/disable permission `P` like in Figure 2. Explain (in a few lines) which other solutions could be used in terms of access control (indicating their advantages and drawbacks).

Q4. If a trusted caller executes method `m1()`, which information could it get about secret buffer `key` ? Assuming that function call `Hash(sum)` returns no confidential information about `sum`, does `m1()` leak some confidential information about `key` ? If yes, which information, if not, why not ?

Exercise 3. ($\sim$ 4 pts).

In most programming languages the lifetime of a local variable of a function cannot exceed this function lifetime. In RUST this rules is enforced at compile time. As an example, in Figure 3, the lifetime of local variable `x` is the one of function `foo`. Hence, when function `main` attempts to write at address `p` an error occurs (since `p` now refers to a non live variable).

Q1. Indicate which vulnerability is avoided by RUST in this example.

Q2. This vulnerability would not be avoided by a similar code written in C. Give a concrete example of gain an attacker may get when exploiting such a vulnerability.

Q2. When using C, is there some solutions to protect a code from this kind of vulnerabilities ?

FIGURE 1 – A still vulnerable C code ...

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define N 128
5
6  void copy_and_process (signed int base, signed int offset, char src[] ) {
7  // copy src[base+offset..N-1] into a local buffer tmp and process it
8  char tmp[N] ;
9  signed int i ;
10
11 if (base<0 || offset <0) {
12  printf("invalid negative argument\n") ;
13         exit(1) ;
14 } ;
15
16 if (base + offset >= N) {
17  printf("use of index larger than the array size ...\n") ;
18         exit(1) ;
19 } ;
20
21 if (base + offset < 0) {
22  printf("use of negative index ...\n") ;
23         exit(1) ;
24 } ;
25
26 for (i=base ; i < base + offset ; i++)
27         tmp[i] = src[i] ;
28 printf("copy is ok, now processing tmp ... \n") ;
29 process (tmp) ; // we do not care about that ...
30 }
31
32 int main() {
33 char *T ;
34 signed int b, o ;
35
36 T = malloc (N) ; // allocate a buffer T of size N
37 scanf("%d", &b) ; // read b from the keyboard
38 scanf("%d", &o) ; // read o from the keyboard
39 copy_and_process (b,o,T) ;
40 }
```

FIGURE 2 – A critical Java class ...

```
#include <stdlib.h>
import java.util.* ;

class C1 {

int key[N] ;  // secret resource of size N

public int m1 (String s, int length) {
 // s is used to authenticate the caller
 int i, sum, result ;
 b = checkAcess(s) ;
 if (b) enablePermission(P) ; // give read acces to buffer key
 try {
   if (b) {
    i=0 ;
    sum= 0 ;
    while (i<length) {
     sum = key[i] + sum ;
     i = i+1 ;
        } ;
     disablePermission(P) ; // disable acccess to buffer key
     if (sum>0)
      result = Hash(sum); // returns a positive hash value
     else
      result = -1 ;
     return result ;
   }
  } catch (IndexOutofBoudsException e) {
   // in case key is accessed out of bounds
  System.out.println("Error !") ;
 }
}
}
```

FIGURE 3 – An example of Rust code ...

```
fn foo () -> &i32 { // foo returns a pointer to a 32 bits integer
  let x:i32 := 0 ; // x is a local variable initialized to 0
  &x ; // returns the address of x
}

fn main ( ) {
 let p = bar() ; // p is initialized with the result of foo
 *p = 42 ; // attempts to store 42 at address p => error !
}
```