**Grenoble INP** **UGA**
**Master CybserSecurity & Master MoSiG** **Year 2019-20**

# Software Security & Secure Programming

## Written Assignment - Wednesday November the 13th, 2019

| **Duration :** 60 minutes – **Authorized documents :** one A4 sheet of paper – answers in English or French |
|---|

## Exercise 1. (~ 8 pts).

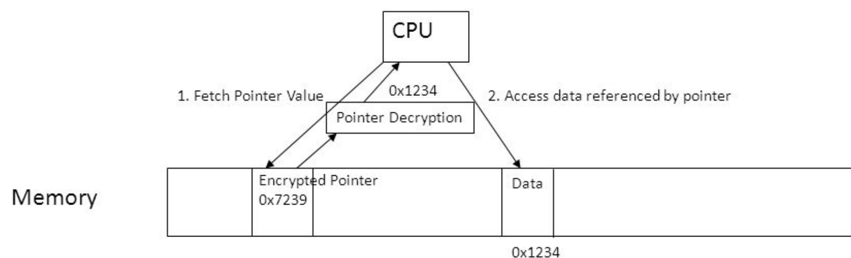
We consider the C program given on Figure 1. It contains a **vulnerability** at lines 29 and 30.

**Q1.** Explain why it is a vulnerability, how an attacker may exploit it, and which gain he/she could get.

**Q2.** Do you think that compiling this program with the `-fstack-protector` option (to tell the compiler to include *canaries* in the stack) is a sufficient protection to avoid this attack ? Which (other ?) kind of protection could be used ? Explain your answers.

**Q3.** In 2010 a U.S. patent was deposed by C. Cowan and its co-authors for a protection called *PointGuard*. This protection consists in ciphering **each pointer** `p` used in a C program with a random key (generated at load-time) using a lightweight cryptographic algorithm (e.g., XOR'ing the pointer value with the key). Then, before each pointer dereference operation (i.e., before each access to `*p`) the pointer value is decrypted at run-time and the resulting address is used (see figure below) :

PointGuard Pointer Dereference




Explain why this protection avoids the attack found in question **Q1**.

**Q4.** Can you see some limitations/drawbacks of this protection mechanism, namely :
examples of vulnerabilities it does not cover, possible ways to overcome this protection by an attacker ?

**Exercise 2. (∼ 8 pts).** We consider the function `BuildKey` of the Java class `Key` given on Figure 2, where :

— the array `key` is supposed to be a **sensible** data, which should not be **corrupted** by any unauthorized user (**integrity property**) ;
— the permission `P` grants *write access* to `key` ;
— other variables are considered as **public** (their content can be corrupted).

**Q1.** Explain the purpose of the `enablePermission()` and `disablePermission()` primitives : why are they useful in this context, since user trustworthiness is checked explicitly at line 12 ?

**Q2.** Function `BuildKey` does not correctly protect the integrity of buffer `key`, i.e., it may happen that a non-trusted user is able to modify it. Explain why.

**Q3.** Assuming variable `passphrase` is a **confidential** (secret) data, indicate which information may *leak* from this data when an **authorized** user executes function `BuildKey`.

## Exercise 3. (∼ 4 pts).

We consider three variants of the C code given on Figure 3 :

— variant1 : **XXX** is replaced by `i<=N`
— variant2 : **XXX** is replaced by `i<=2*N`
— variant3 : **XXX** is replaced by `i<=N+10`

We compile these 3 variants with the `-fstack-protector` option (i.e., telling the compiler to add *canaries* inside the stack) and we observe the following results when running them :

— variant1 produces no error (and no output)
— variant2 produces a `segmentation fault` error message (without any other information)
— variant3 produces a `*** stack smashing detected ***` error message

Explain each of these results (you can draw the execution stack to justify your answer).

Figure 1 – A vulnerable C code ...

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int foo () { printf("foo\n") ; return 1 ; }
5
6  int bar () { printf("bar\n") ; return 1 ; }
7
8
9  // declaration of a type called "Object" as a 2-fields structure
10 typedef struct {
11   char buf[20] ;    // field buf = buffer of 20 chars
12   int (*func)() ;   // field func = function pointer
13 } Object ;
14
15 int main() {
16
17   Object *o1, *o2 ;
18
19   o1 = (Object *) malloc (sizeof(Object)) ;  // allocates an Object o1
20   o2 = (Object *) malloc (sizeof(Object)) ; // allocates an Object o2
21   if (o1==NULL || o2==NULL)
22     return -1 ; // terminates if one of the allocations failed
23
24   // initializes fields func of o1 and o2 with function addresses
25   o1->func = &foo ;
26   o2->func = &bar ;
27
28   // initializes fields buf of o1 and o2 with user inputs
29   scanf("%s", o1->buf) ;
30   scanf("%s", o2->buf) ;
31
32   // calls the function pointed to by fields func of o1 and o2
33   (*(o1->func))() ;  // supposed to call foo
34   (*(o2->func))() ;  // supposed to call bar
35   return 0 ;
36 }
```

FIGURE 2 – A critical Java class . . .

```
1  class Key {
2
3    int nbSpecials = 0 ;
4    int key[] ;
5
6  int BuildKey (String user, String passphrase) {
7
8    int size=passphrase.length();
9    int j=0;
10
11   for (int i=0; i<size ; i++) {
12     if (isTrusted(user)) {  // if the user is trusted
13        enablePermission(P) ; // give write access to buffer key
14        try {
15          if (isAscii(passphrase.atChar(i)) {
16                  // check if current char is an Ascii char
17              key[j] = f1(passphrase[i]) ;
18              j = j+1 ;
19          } else {
20              key[j] = f2(passphrase[i]) ;
21              nbSpecials = nbSpecials +1 ;
22          };
23          disablePermission(P) ; // remove write access to buffer key
24        } catch (ArrayIndexOutofBoundsException e) {
25               ... // handle buffer overflow error}
26     }
27   }
28   return nbSpecials ;
29 }
30 ...
31 }
```

FIGURE 3 – An example of C code . . .

```
1  #include <stdio.h>
2
3  #define N 128
4
5  int main() {
6  int buff[N] ;  // defines an array of int with indexes ranging from 0 to N-1
7
8  int i ;
9  for (i=0 ; XXX ;   i++)
10     buff[i] = 42 ;
11 return 0 ;
12 }
```