Software Security & Secure Programming

Security weaknesses in programming languages - Exercises

Exercise 1.

In C, signed integer overflow is undefined behavior. As a result, a compiler may assume that signed operations do not overflow. The code below is supposed to provide sanity checks in order to return an error code when the `offset + len` does overflow :

```
int offset, len ;  // signed integers
...
/* first check that both offset and len are positives */
if (offset < 0 || len <= 0)
    return -EINVAL;
/* if offset + len exceeds the MAXSIZE threshold, or in case of overflow,
    return an error code */
if ((offset + len > MAXSIZE) || (offset + len < 0)
    return -EFBIG // offset + len does overflow
/* assume from now on that len + offset did not overflow ... */
```

Explain why this code is vulnerable (i.e., the checks may fail). Propose a solution to correct it.

Exercise 2.

In C, dereferencing a `NULL` pointer is undefined behavior. As a result, a compiler may assume that all dereferenced pointers are non-null. The `GCC` compiler choose memory address 0 for the `NULL` value. Depending on the processor, address 0 may be mapped to a valid memory page.

Explain what may happen in this case when running the code fragment below :

```
struct my_struct *s = f();
int t = s-> f ;   // s is dereferenced
if (!s)
return ERROR;
...
```

How to correct this code ?

Exercise 3.

Let us consider the following C function :

```
void func(unsigned int nb, int tab[]) {
```

```
    int *dst;
    dst = (int *) malloc(sizeof(unsigned int)*nb);
    for (int i=0; i <nb; i++)
        dst[i]=tab[i] ;
}
```

Q1. List all the *runtime errors* that could occur when executing this function, explaining *how* they can occur (i.e., under which conditions).

Q2. Explain why this function is *vulnerable*.

Q3. Propose a solution to make this function *secure*.

Q4. What is strange in the code below, how to correct it ?

```
#define M -1

int main() {
   unsigned int ui ;
   scanf("%u", &ui) ; // read a value for ui
   while (ui > M) {
      ... //
      scanf("%u", &ui) ; // read another value for ui
   } ;
   return 0 ;
}
```

Exercise 4.

Here is an excerpt of the CERT coding standarts [1] regarding operations on unsigned integer (Rule INT3–C) :

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Q1. According to the CERT, this "wrap-around" behavior should be avoided (at least !) in the following situations :

— integer operand on any pointer arithmetic, including array indexing
— assignement expressions for the declaration of a variable length array

Give some "security crtitical" examples for each of these situations.

Q2. Here is code fragment extracted from OpenSSH 3.3 :

```
unsigned int i, nrep;  // user inputs
...
nrep = packet_get_int() ;
response = malloc(nrep*sizeof(char*));
if (response != NULL)
   for (i=0; i<nrep; i++)
       response[i] = packet_get_string(NULL)
...
```

1. https://www.securecoding.cert.org

Explain why this code is vulnerable, giving the corresponding inputs. Propose a (general) way to correct it.

Exercise 5.

The `safewrite` function below is supposed to check for out-of-bounds when accessing an array.

```
void safewrite (int tab[], int size, signed char ind, int val) {
   if (ind<size)
       tab[ind]=val;
   else
       printf("Out of bounds\n");
}
```

However, this check may fail in one on the two following calls to this function :

```
int main() {
unsigned int size=120 ;
int tab[size];
safewrite(tab, size, 127, 0);
safewrite(tab, size, 128, 1);
return 0;
}
```

Can you tell which one, why it fails and how to strenghten the `safewrite` function ?

Exercise 6.

Let us consider the C code below :

```
void main ()
{
  char t;
  char t1[8] ;
  char t2[16] ;
  int i;
  t = 0;
  for (i=0;i<15;i++) t2[i]=2;
  t2[15]='\0' ;
  strcpy(t1, t2) ;
  printf("La valeur de t : %d \n", t);
}
```

The *stack layout* (i.e., the way local variables are stored in the stack) may vary from one compiler to another. Draw a stack layout coresponding to each of these situations :

— the program prints 2 as the value of `t`
— the program crashes
— no crash, and the program prints 0 as the value of `t`