Ensimag **UGA**
Master CybserSecurity & Master MoSiG

Software Security & Secure Programming

Security weaknesses in programming languages - Exercises

Exercise 1 C - Arithmetic overflow (unsigned integers)

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 expression `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 ... */
```

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

Exercise 2 C - A vulnerable code

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] ;
}
```

1. List all the *runtime errors* that could occur when executing this function, explaining *how* they can occur (i.e., under which conditions).
2. Explain why this function is *vulnerable*.
3. Propose a solution to make this function *secure*.

Exercise 3 C - Arithmetic overflows (signed integers)

Here is an excerpt of the CERT coding standards [1] regarding operations on *unsigned integers* (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.

1. 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
   — assignment expressions for the declaration of a variable length array
   Give some "security critical" examples for each of these situations.

2. 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)
   ...
   ```

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

Exercise 4 C - Buffer overflow

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

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

1. Can you tell which one, and why it fails ?

2. How to strengthen the `safewrite` function ?

Exercise 5 C - Buffer overflow

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 corresponding 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`

Exercise 6 C - Dynamic allocation

We consider the following C code :

```
typedef struct {void (*f)(void);} st;
void nothing (){ printf("Nothing\n"); }

int main(int argc , char * argv [])
{ st *p1;
 char *p2;
 p1=(st*) malloc(sizeof(st));
 p1 ->f=&nothing;
 free(p1);
 p2=malloc(strlen(argv [1]));
 strcpy(p2 ,argv [1]);
 p1 ->f();
 return 0;
}
```

1. Explain why this program contains an *undefined behavior*, and how it can be exploited.

2. A programming advice is to assign pointers to `NULL` as soon as they are freed.

a) Explain why this solution may help, and apply this technique to previous program.

b) Explain why this solution may fail when using an optimizing compiler.

c) Give an example showing that this solution is not *complete*.

d) Which kind of code analysis may be used to get a complete solution ?

Exercise 7 PHP - Vulnerable code

This PHP code snippet intends to take the name of a user and list the contents of that user's home directory.

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Explain why this code is not secure and propose a correction.

Exercise 8 Python & PHP - Vulnerable code

We consider Python programs able to create directories using the `mkdir` path command

`os.mkdir(path[, mode])` : Create a directory named path with numeric mode mode. The default mode is 0777 (octal). If the directory already exists, exception `OSError` is raised.

1. Give a list of the potential vulnerabilities associated to this command.

2. Consider the Python program below. Explain what it does. The security rule telling to *raise privileges only in the code parts it is necessary* may be violated in this example. Propose a correction.

```
def makeNewUserDir(username):
  if invalidUsername(username):
  #avoid CWE-22 and CWE-78
    print('Usernames cannot contain invalid characters')
    return False
  try:
    raisePrivileges()
    os.mkdir('/home/' + username)
    lowerPrivileges()
  except OSError:
     print('Unable to create new user directory for user:' + username)
     return False
  return True
```

3. We consider the following PHP code. Explain why it is insecure and how to correct it.

```
function createUserDir($username){
  $path = '/home/'.$username;
  if(!mkdir($path)){ return false;}
  if(!chown($path,$username)){rmdir($path); return false;}
  return true;}
```

Exercise 9 C - Erase sensitive data

A good secure coding rule is to erase sensitive data (see chapter 13 of "Secure Programming Cookbook for C and C++", by by John Viega with Rakuten Kobo). Let's consider the following program :

```
int get_and_verify_password(char *real_password) {
  int  result;
  char *user_password[64];
  get_password_from_user_somehow(user_password, sizeof(user_password));
  result = !strcmp(user_password, real_password);
  memset(user_password, 0, strlen(user_password));
  return result;
}
```

Explain the weaknesses of this code. Can we alway guarantee that `user_password` will be erased ? I no, how to correct that ?

Exercise 10 C - Secured `string` functions

Secure programming rules advice to not use insecure library functions like `strcat`, `strcpy`, ..., but to use their secure versions instead (see for instance `https://msdn.microsoft.com/en-us/library/bb288454.aspx`).

Another good advice is to systematically check error codes returned by these functions. For example : `errno_t strcpy_s( char * restrict s1, rsize_t s1max, const char * restrict s2);` behaves as follows :

> The `strcpy_s()` function succeeds only when the source string can be fully copied to the destination without overflowing the destination buffer. Specifically, the following checks are made : The source and destination pointers are checked to see if they are NULL. The maximum length of the destination buffer is checked to see if it is equal to 0, greater than RSIZE_MAX, or less than or equal to the length of the source string. Copying is not allowed between objects that overlap. When a runtime-constraint violation is detected, the destination string is set to the null string (as long as it is not a null pointer, and the maximum length of the destination buffer is greater than 0 and not greater than RSIZE_MAX), and the function returns a nonzero value.

1. Indicate where the following code is vulnerable :

   ```
   void complain(const char *msg) {
     static const char prefix[] = "Error: ";
     static const char suffix[] = "\n";
     char buf[BUFSIZ];

     strcpy(buf, prefix);
     strcat(buf, msg);
     strcat(buf, suffix);
     fputs(buf, stderr);
   }
   ```

2. Propose a more secure version of this code using the suggested secure library functions.

3. We would like to verify that error codes returned by library functions are systematically checked in a program. Which kind of solution can we suggest ? What are the difficulties for such a solution ?

4. The C Standard `strncpy()` function is frequently recommended as an alternative to the `strcpy()` function. What are the possible difficulties when using this function ?

Exercise 11   C - NULL pointer dereference

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 current memory mapping (OS version, previous calls to the `mmap()` function, etc.) address 0 may be mapped to a *valid* memory page (i.e., no crash when it is accessed).

Under these hypotheses, explain why this (realistic !) code fragment is *insecure* :

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

How to correct this code ?

Exercise 12   C - Implicit conversions

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 ;
  }
```