Ensimag **UGA**
Master CybserSecurity

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) when compiled with optimization options.

2. Propose a solution to correct it.

Exercise 2 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) ;   // copy t2 into t1
  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 :

(a) the program prints 2 as the value of `t`

(b) the program crashes (because of an invalid memory access)

(c) no crash, and the program prints 0 as the value of `t`

Exercise 3 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 4 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 5 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 6 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;
}
```

What is the purpose of the call to `memset` ? Why this solution could be unsufficient ? How to improve it ?

Exercise 7 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.

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

Exercise 8 C - Implicit conversions

The following C function calls the standard library function `read` those profile is

    size_t read(int fd, void *buf, size_t count)[2]

which is supposed to read `count` bytes from the file `fd` to the buffer `buf`.

```
int read_user_data(int sockfd) {
  int length, sockfd, n;
  char buffer[1024];

  length = get_user_length(sockfd);
  if(length > 1024){
    error("illegal input, not enough room in buffer\n");
    return 1;
  }

  if(read(sockfd, buffer, length) < 0){
    error("read: %m");
    return 1;
  }

  return 0;
}
```

Explain why this function is vulnerable, and how to correct it …

[2] with `size_t` defined as an `unsigned int`