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.

Answer of Exercise 1

1. — after the first check, both len and offset are assumed to be positive
   — since signed integer overflow is undefined behavior, the compiler then assumes that `offset + len` cannot be < 0 and removes this check ...

   As a result if len and offset are large integers, the whole sanity check is by-passed ...

2. Possible solutions ?
   1. try to modify the ordering of the checks, e.g.,

      ```
      if ((offset + len > MAXSIZE) || ((offset + len < 0) && offset > 0 && len >= 0))
          return -EFBIG // offset + len does overflow
      if (offset < 0 || len <= 0)
          return -EINVAL;
      ```

      Not so easy ...
   2. see also the solution proposed by the CERT (`https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow`) :

      ```
      if (((len > 0) && (offset > (INT_MAX - len))) ||
            ((len < 0) && (offset < (INT_MIN - len)))) {
          /* Handle error */
        } else {
          sum = offset + len;
        }
      ```
   3. convert offset and len into unsigned values (which enforces wrap-around in case on overflow ...)
   4. With the GCC compiler : use the "-fno-strict-overflow" option to tell the compiler to not consider such overflows as undefined behavior, and use the "-fwrapv" option to "wrap around" in case of overflows (as in Java)

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`

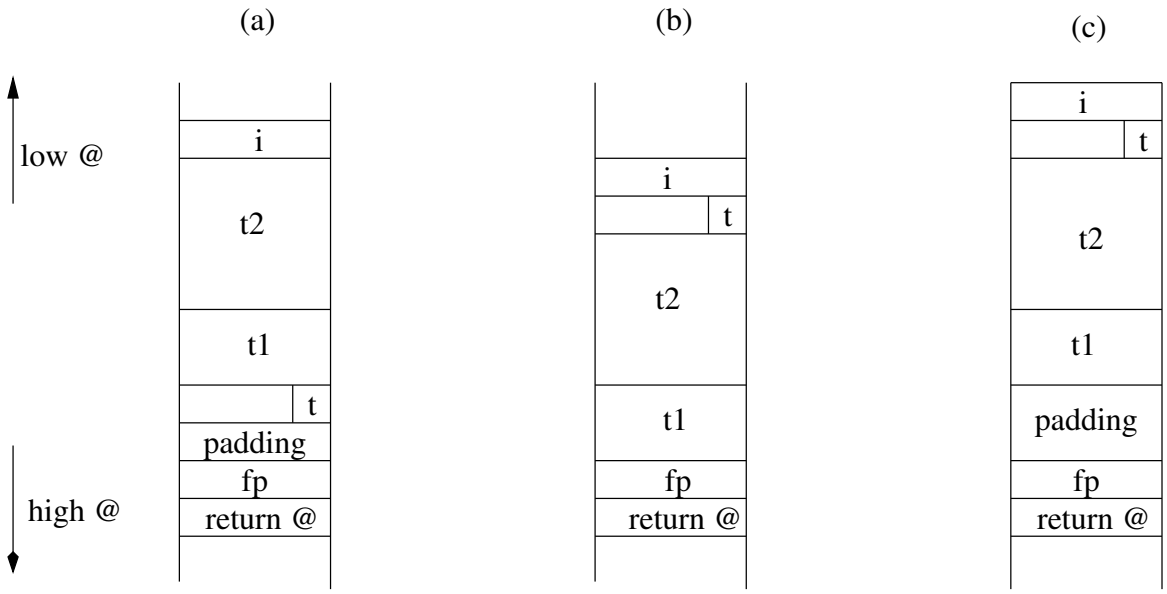
Answer of Exercise 2

When t2 is copied into t1 then t1 overflows and the value "2" may :

(a) either overwrite t

(b) or overwrite the frame pointer (fp) and/or the return address (leading to a crash)

(c) or overwrite some padding zone (with no influence on the program behavior).



The picture above gives the stack layout according to these three situations.

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 ?

Answer of Exercise 3

1. Pointer `p1` is freed and then accessed later on. Accessing a non-allocated memory zone is undefined behavior. In this example this results in a *use-after-free* : the 2nd call to `malloc` may re-use the memory zone just freed, which is then overwritten with user supplied data (`strcpy(p2 ,argv [1]);`). Hence, the call to `p1 ->f();` may result in **arbitrary code execution** ...
   Rk : Note that a *buffer overflow* may also occur if the executable is ran without arguments (i.e., `argv[1]` is not defined), but this is likely to lead to a (non-exploitable) crash ...

2. a) The call `p1 ->f();` then dereferences a NULL pointer, which (usually !) leads to a crash ...If we assign pointers to NULL when they are freed, then we can check them before any dereference operation (if the pointer is NULL then we can raise an error) :

```
int main(int argc , char * argv [])
{ st *p1;
 char *p2;
 p1=(st*) malloc(sizeof(st));
 p1 ->f=&nothing;
 free(p1);
 p1==NULL ;
 p2=malloc(strlen(argv [1]));
 strcpy(p2 ,argv [1]);
 p1 ->f();  // should produce a crash here ...
 return 0;
```

   b) Dereferencing a NULL pointer is an undefined behavior as well, and an optimizing compiler may assume that when a pointer is dereferenced then it is non-NULL. Moreover, dereferencing a NULL pointer may not always result in a crash ...(see exercise **??**).

   c) Assigning NULL to a freed pointer does not change its aliases :

```
p1 = malloc(...) ;
p2 = p1 ;
free (p1) ;
p1 = NULL ;
/* now p2 is freed, but not NULL ! */
... p2-> ... /* use-after-free */
```

   Hence this solution is not complete (it does not prevent all errors ...)

   d) Making this solution *complete* would require to identify *all* pointer aliases. This analysis is not *decidable* (only approximated alias analysis are ...).

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.

Answer of Exercise 4

This code is subject to the first variant of OS command injection (CWE-88) :

```
;rm -rf /
```

Which would result in `$command` being :

```
ls -l /home/;rm -rf /
```

Since the semi-colon is a command separator in Unix, the OS would first execute the `ls` command, then the `rm` command, deleting the entire file system.

To correct this code you should *escape inputs correctly*. Injection vulnerabilities occur when untrusted input is not sanitized correctly. If you use shell commands, be sure to scrub input values for potentially malicious characters (like `;` `&` or `|`). You can use for instance predefined functions like `escapeshellcommands` or `escapeshellarguments` (see for instance the on-line PHP manual `http://php.net/docs.php`). Also note that this example code is vulnerable to Path Traversal (CWE-22) and Untrusted Search Path (CWE-426) attacks.

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;}
```

Answer of Exercise 5

1. The use of this `mkdir` command may lead to several attack : Command injection (CWE-77), Argument injection or modification (CWE-88), Path Traversal (CWE-22) and Untrusted Search Path (CWE-426).

2. While the program only raises its privilege level to create the folder and immediately lowers it again, if the call to os.mkdir() throws an exception, the call to lowerPrivileges() will not occur. As a result, the program is indefinitely operating in a raised privilege state, possibly allowing further exploitation to occur. A possible way to correct it is to call `lowerPrivileges()` at the beginning of the exception handler ....

3. Because the optional "mode" argument is omitted from the call to mkdir(), the directory is created with the default permissions 0777. Simply setting the new user as the owner of the directory does not explicitly change the permissions of the directory, leaving it with the default. This default allows any user to read and write to the directory, allowing an attack on the user's files. The code also fails to change the owner group of the directory, which may result in access by unexpected groups.

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 ?

Answer of Exercise 6

`user_passord` is not *used* after the `strcmp` (it is a **dead** variable at this point).
Hence, the optimizer may suppress the call to `memset` ...
A solution is to declare `user_password` as *volatile*, but it may not be sufficient, the compile optimization may still remove the call to `memset` (since it knows the behavior of this standard function). A more secure option is then to write `memset` by hand.

If you're writing code for Windows using the latest Platform SDK, you can use `SecureZeroMemory( )` instead of `spc_memset()` to zero memory. `SecureZeroMemory()` is actually implemented as a macro to `RtlSecureMemory()`, which is implemented as an inline function in the same way that `spc_memset()` is implemented, except that it only allows a buffer to be filled with zero bytes instead of a value of the caller's choosing, as `spc_memset()` does.

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

Answer of Exercise 7

1. pointer arithmetic for array indexing :

   ```
   tab[i + off] ;
   ```

   variable length array :

   ```
   size = s1 + s2 ; tab = (int *)malloc(size) ;
   ```

2. The size of the malloc call can overflow ... (and become 0, or much smaller than what is expected !). Then a buffer overflow may occur ...

   A possible correction :

   ```
   nrep = packet_get_int() ;
   if (nrep > UINT_MAX /sizeof(char*))
   {printf("Overflow"); exit(0);}
    else {
      ...
    }
   ```

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`

Answer of Exercise 8

The user-supplied value `length` is declared as a (signed) integer and hence converted into an unsigned one when calling `read`. Moreover, the check `(length > 1024)` will always return true when `length` is negative. As a result, a malicious user may trigger a buffer overflow when calling `read`, possibly overwritting its return address.

The easiest way to correct this issue is probably to declare `length` as an **unsigned** integer ...