Lecture 23: Memory Safety

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/23-memory-safety.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we discuss a problem that is inherent to C: the fact that the language is not **memory-safe**.

Memory Unsafety in C Programs

C is not memory-safe. This means that there is no protection against a range of bugs when dealing with memory accesses. These bugs can relate to **spatial safety**: in C the programmer is free to read or write anywhere he/she wants to in memory. Bugs in this category include out of bound array accesses and bad pointer dereference (e.g. pointer being `NULL` or mistakes in pointer arithmetics). Other bugs relate to **temporal safety**: the compiler cannot check things like accessing a dynamically-allocated buffer after it has been freed: this is a use-after-free. These bugs lead to **undefined behaviour**: it can be a crash if the memory accessed is not mapped (remember the virtual address space is overall scarce), or incoherent (and hard to debug) behaviour when the memory accessed is mapped but does not contain what the programmer expects.

C is not memory safe for various reasons, including performance: adding runtime checks against these bugs is in theory possible, but it hurts performance. There is no bound check on buffers and array accesses. Uninitialised variables generally contain random garbage. Dynamic memory allocation can lead to a lot of mistakes that won't be caught by the compiler. Such as memory leaks, double free, use-after-free, and so on. All these issues are hard to detect and to debug. They can also be hard to reproduce: sometimes the bug only manifest after a long time, several executions, or on a particular platform.

An important thing to note is that, in addition to crashes, which in some sense are a good outcome because they indicate something needs to be fixed, these memory issues can have dramatic implication in terms of security. Here we will study 4 example of memory bugs that lead to security issues.

Example 1: Infoleak

The first example regards the leak of confidential information. Consider this program:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *welcome_message = "Hi there! How is it going?\n"; // 27 char
char *password = "secret";
char entered_password[128];

int main(int argc, char **argv) {
    // Print welcome message character by character
    for(int i=0; i<27; i++) {
        printf("%c", welcome_message[i]);
    }

    printf("Please input the password:\n");
    scanf("%s", entered_password);

    if(!strcmp(entered_password, password)) {
        printf("Passowrd ok!\n");
    /* ... */
    } else {
        printf("Wrong password! aborting\n");
    }

    return 0;
}
```

The program checks a platform entered by the user using `scanf`. It starts by printing a welcome message character by character. Then asks for a password from the user and compares the input to the correct password. If the password is correct, the program then goes on to do something important. The program is distributed in binary-only form, so unauthorised users do not have access to the sources and cannot see the password.

Let's assume that during an update, the welcome message is updated as follows:

```c
char *welcome_message = "Hi there!\n"; // shortened welcome message, only 11 chars now
```

Let's assume that the programmer forgets to update the number of iteration of the `for` loop printing that message. The message being now much smaller, the printing loop will overflow the `welcome_message` string and print whatever bytes are present next in memory. It may just be garbage ... or not: due to how the compiler lays out variables in memory, there are actually good chances that the password is very close to the welcome message in memory. The password is in fact located right after it. So the printing loop overflows the welcome message and reads bytes from password. And the password is leaked to the standard output. Running the faulty program gives:

```
$ ./infoleak-updated
Hi there!
secretPlease inPlease input the password:
```

Example 2: Sensitive Data Tampering

Let's see a second example in which we tamper with a password to bypass it. Consider this program:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char user_input[32] = "00000000000";
char password[32] = "secret";

int main(int argc, char **argv) {

    if(argc != 2) {
        printf("Usage: %s <password>\n", argv[0]);
        return 0;
    }

    strcpy(user_input, argv[1]); // oopsie!

    if(!strncmp(password, user_input, strlen(password))) {
        printf("login success!\n");
    } else {
        printf("wrong password!\n");
    }

    /* ... */
    return 0;
}
```

It takes the password as command line parameter. The password is copied in a buffer `user_input`. The content of that buffer is compared with the correct password, and if they match the program continues. The issue is that there are no checks done by `strcpy` regarding the sizes of the source and destination strings.

In memory, it is very likely that the compiler will place the correct password right after the `user_input` buffer. If the user passes a string as command line parameter which size is larger than the 32 bytes of that buffer, `strcpy` will overflow `user_input` and start overwriting the password. The user hence has the capacity to rewrite the correct password with the value of his/her choice. That value should be the same as the first 25 bytes that will go into `user_input`, so that when `strcmp` is called the strings will match:

```
$ ./tampering xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
login success!
```

Example 3: Stack Smashing

Stack smashing is an old attack where a buffer overflow can be exploited on the stack to take over the control flow of an application. It was originally presented in 1996[1]. Before going over the attack with an example, we first have to understand how the machine uses the stack to manage function call and return operations.

Functions from the Machine Code Point of View

The stack is a dynamic data structure in memory. It holds, amongst other things, local variables as well as function parameters and return values. The space on the stack dedicated to a function's data (local variables, etc.) is contiguous in memory, it is called the function's stack frame. When a function is called, the stack's size is increased to create the corresponding frame. In addition to the function's data, its frame also contains the **return address**. This is the address in the program's code (that is present somewhere else in memory) to which the CPU will jump once the function returns. The central idea of stack smashing is to overwrite the return address of a function `f` with the address of code the attacker wishes to execute (e.g. the successful branch of a password check): when `f` returns, rather than returning to `f`'s caller, the program jumps to the target code.

Target Program

Consider this program:

```
char *password = "super-secret-password";
void security_critical_function() {
    printf("launching nukes!!\n");
}
void preprocess_input(char *string) {
    char local_buffer[16];
    strcpy(local_buffer, string);
    /* work on local buffer ... */
}
int main(int argc, char **argv) {
    if (argc != 2) { /* ... */ }
    preprocess_input(argv[1]);
    if(!strncmp(password, argv[1],
            strlen(password)))
        security_critical_function();
    else
        printf("Unauthorised user!\n");
```

[1] http://www.phrack.org/archives/issues/49/14.txt

```
    return 0;
}
```

This program takes a password as command line parameter. The input is passed through a function that preprocess it after having it copied in a local buffer with `strcpy`. And if the password is correct we execute a security critical function. There is a bug in this program: `strcpy` check for the size of `string` prior to copying it in `local_buffer`, letting the attacker overflow that buffer if data passed on the command line is larger than 16 bytes. `local_buffer` is a local variable, hence it is allocated on the stack. Similarly to the other examples, assume an attacker that only has access to the program's binary, and that does not know the password.

The Attack

With stack smashing we will use this overflow to overwrite the return address of `preprocess_input` with the address of `security_critical_function` in order to bypass the password check. This is possible because on x86-64 the stack grows down, from higher to lower addresses. The return address is located after `local_buffer` on the stack, so overflowing it with `strcpy` allows overwriting the return address with whatever the attacker inputs from the command line. The process is illustrated on this schema:

```
           +----------------------+
         | |                      |High addresses
         | |  main's stack frame  |
         | |                      |
         | +----------------------+ ^
         | |  preprocess_input's  | |
         | |    return address    | |
         | +----------------------+ |Overflow
    Stack| |                      | |with
   growth| +----------------------+ |strcpy
direction| |                      | |
         | |     local_buffer     | |
         | |                      | |
         | +----------------------+ |
         | |                      |
         | |                      |
         | |                      |Low addresses
         v +----------------------+
```

The schema represents the state of the stack at the time `preprocess_input` runs and `strcpy` is called. We have the calling context (`main`) local variables first. Then the return address that was pushed when we called `preprocess_input`. And then `preprocess_input`'s frame with its local variables including `local_buffer`. Remember that because the stack grows down, lower addresses are on the bottom of this graph, so when we overflow `local_buffer` we are effectively writing up the top of the stack towards the return address By using a carefully crafted input string, we can overwrite the return address with the address of a function we would like to execute. For example the address of `security_critical_function`. And when the execution returns from `preprocess_input`, the CPU will jump to `security_critical_function` rather than returning to `main`: the password check is bypassed.

See the complete program's sources[2] for instructions on how to reproduce this attack. On the computer this code was tested, the payload (injected with `echo -e` and `xargs` to produce bytes and not ascii characters) looks like this:

```
$ echo -e "\x11\x11\x11\... (24 bytes of \x11 padding) ... \x55\x16\x40\x00\x00\x00\x00\x00" \
    | xargs --null -t -n1 ./stack-smashing
./stack-smashing ''$'\021\021\021\021\021\021\021\021\021\021\021\021\021\021\021\021\021\021
    \021\021\021\021\021\021''U'$'\026''@'
launching nukes!!
xargs: ./stack-smashing: terminated by signal 11
```

[2] https://github.com/olivierpierre/comp26020-devcontainer/blob/master/23-memory-safety/stack-smashing.c

Example 4: Use-After-Free

A use-after-free happens when the programmer mistakenly uses an object after it has been freed. Consider this program:

```c
typedef struct {
    double member1;
    double member2;
    void (*member3)(int);
} my_struct;

void print_hello(int x) {
    printf("Hello, parameter: %d\n", x);
}

void security_critical_function() {
    printf("Launching nukes!\n");
    /* ... */
}

int main(int argc, char **argv) {

    /* allocate and init ms */
    my_struct *ms = malloc(sizeof(my_struct));
    ms->member1 = 42.0; ms->member2 = 42.0;
    ms->member3 = &print_hello;

    /* call the function pointer */
    ms->member3(12);

    free(ms);

    char *buffer = malloc(12);
    strcpy(buffer, argv[1]);

    ms->member3(12); // Oopsie!
    exit(0);
}
```

Here `ms` is allocated with malloc, then freed, then mistakenly used right before the call to `exit(0)`. This issue is not caught by the compiler, and in many scenarios will not manifest either at runtime.

If we look at the data structure declaration, it has an int member and a second member that is a **function pointer**. A function pointer is a variable that can store the address of a function, see how it is assigned in `main`: `ms->member3 = &print_hello;`. We put the address of the print_hello function in the member. And this function can be called through the function pointer, as it also done later: `ms->member3(12);`. Between the moment what is pointed by `ms` is freed and the use-after-free, a buffer pointed by `buffer` is allocated with `malloc` and a command line parameter is copied in that buffer with `strcpy`. Note the lack of check on the size of the command line argument: once again the attacker can overflow the memory pointed by `buffer`.

Let's see how we can exploit use this program to force the execution of security_critical_function, that is not supposed to be called in this particular program. The central idea behind many attacks leveraging a use-after-free vulnerability is to replace the data structure being used after free with a payload that will corrupt the behaviour of the program once the use-after-free happens. `buffer`, as well as what is pointed by `ms`, are allocated on the heap. `malloc` manages allocations on the heap, and to avoid memory waste `malloc` will reuse freed memory to serve new allocation requests: in other words, it is very likely that the memory used to hold what was pointed was `ms` will be reused for `buffer`. Recall that through the buffer overflow, the attacker has control over the content of `buffer`. The attacker is going to write inside `buffer` the address of `security_critical_function`, at the precise location where, if interpreted as an `ms` data structure, the function pointer would be located. Once the function pointer is dereferenced, the CPU will in effect jump to `security_critical_function`. Please see the full source code[3] for instruction on how to reproduce that attack.

On the computer this code was tested, the payload looks like that:

```
echo -e "\x11\x11\x11\ ... (16 bytes of \x11 padding) ... \x7c\x16\x40\x00\x00\x00\x00\x00" \
    | xargs --null -t -n1 ./use-after-free
./use-after-free ''$'\021\021\021\021\021\021\021\021\021\021\021\021\021\021\021
    \021''|'$'\026''@'
Hello, parameter: 12
Launching nukes!
# program continue to misbehave after that
```

Conclusion

C is not memory safe. The memory safety bugs introduced by programming mistakes lead not only crashes, but more importantly to **security vulnerabilities** that can have very serious consequences.

[3]https://github.com/olivierpierre/comp26020-devcontainer/blob/master/23-memory-safety/use-after-free.c