Analysing Software Security Vulnerabilities

Luigi Coniglio - s1987046

December 5, 2017

B: Writing secure code

In this section I describe my approach and the results obtained throughout the process of securing the program *sint*.

Part 1. briefly describes how the bugs found and other decisions regarding the implementation are documented. Part 2. illustrates my approach: the tools used, the order of their usage, etc. . Finally, in part 3. I describe the results and highlight some of the most notable bugs.

1. Documentation

For ease of correction, and in order to show that the bugs were indeed understood, all the findings and are documented on top of each function. [1]

For instance, every bug found inside a function is properly documented in the *BUGS FIXED* section which can be found on top of the function declaration in the *.c file (just after the specifications, if any). Furthermore a section *OBSERVATIONS* is occasionally used to describe some minor bugs or design considerations. In general the code is heavily commented and I tried to make it as readable as possible.

Additionally I decided to include a list of the bugs that have been found using external tools. For each bug this list contains the name and command line option of the tool that was able to discover the bug and the tool's output. In some cases, (e.g. executions using *valgrind*) the input of the program is also given. Findings are ordered chronologically (the first in the list is also the first bug found). This list can be found in the file *list_tools_findings*.

2. Approach

The approach I decided to adopt for this assignment consisted of trying to reduce the number of bugs using automatic tools first for later proceeding with the manual review of the code.

[1] This is particularly true for the function *read_data* which, in order to avoid the unsafe behavior of *sscanf* when parsing numeric values, has been almost completely reimplemented (using the safest function *strtol* in order to parse the user-provided integer).

Table 1: Programs versions

Program	**Version**
cppcheck	1.61
valgrind	3.10.1
gcc	4.8.4

I decided to take this approach in order to learn more about the potentials and limitations of the automatic tools and take maximum advantage from their usage (minimizing the number of bugs left before inspecting the code manually). Indeed, thanks to this approach, I was able to identify what kind of bugs are particularly resilient to automated analysis techniques (those are design flaws most of all). The list below shows which tools/actions have been used/taken in the respective order:

1. Compiler warnings
2. Static analysis using *cppcheck*
3. Dynamic analysis using *valgring* (with memcheck)
4. Dynamic analysis using *AddressSanitizer*
5. Manual review

Static analysis tools have a great advantage in usability if compared to dynamic analysis tools, since they can be used even previous to compilation. This is the main reason why, as the above list shows, *cppcheck* has been used before *valgrind*.

As shown in the previous list, I decided to explore a little bit further the panorama of dynamic analysis tools by using an additional tool: Google's *AddressSanitizer* (aka ASan). Unfortunately *ASan* wasn't of much help for this part of the assignment since *valgrind* was already able to catch already the bugs that *ASan* could have found. Nonetheless *ASan* has proven to be an excellent tool for the analysis of the more complex applications covered in part C (partially for its ease of use due to its outstanding performances in terms of execution speed if compared to *valgrind*).[2]

Table 1 show the version of the tools used for this assignment.[3]

In addition I created extra tests in order to address some particular states of the program. And, regarding the design choices related to the development of the required functions, I tried to follow the safest possible approach (for example, avoiding recursive constructs and thus stack overflows).

[2]Indeed *valgrind* aims to be an heavyweight binary analysis tool. For this reason it performs in-depth analysis involving a larger amount of runtime checks if compared to other binary analysis tools. Furthermore *valgrind* uses JIT instrumentation (with VEX instructions as intermediate representation). Those characteristic entails a significant slowdown of 20x in the case of *memcheck*, compared to the 2-3x slowdown of *ASan* [1],[2]

[3]Note that *ASan* has been embedded in *gcc* starting from version 4.8

3. Results

Table 2 shows, for each function of the program, the list of bugs found and fixed (23 in total). Since, for reason of space, it is not be possible to discuss each single bug found in the context of this report, I decided, as already said in part 1., to include a description of each bug in the comments on top of each function in the source code. Furthermore, for the content of this report, I decided to focus on some of the most common bugs found and some the bugs which could have the most severe security implications

Table 2: List of bugs found and fixed

Functions	**Bugs fixed**
read_data	Stack-based buffer overflow Potential uninitialized values Potential integer overflow (or underflow) Negative ages were allowed Improper use of *sscanf* to parse integers
handle_command	Format string bug Return value of read_data not checked for errors Return value of read_data not freed
read_command	NULL pointer dereferencing (malloc) Memory leak if EOF was encountered Integer overflow Memory leak if realloc fails NULL pointer dereferencing (realloc) Invalid reads/writes on the heap Invalid write on the heap (off-by-one style bug) Command compromised if EOF is encountered while reading Unlimited amount of bytes read User input was dropped in some cases
main	Memory leak NULL pointer dereferencing
data_compare	Specification mismatch
data_new	NULL pointer dereferencing
sortedcontainer_new	NULL pointer dereferencing

NULL pointer dereferencing
This was the most common bug found in the original code of the program. This problem was caused most of the time by the use of pointers without any previous validation. The code above, taken from the function *sortedcontainer_new* shows an example:

```
sortedcontainer* d = malloc(sizeof(sortedcontainer));
d->root = NULL;
```

Here the pointer *d* is used without first checking if *malloc* was able to successfully allocate some memory (i.e. *malloc* could have returned a NULL pointer), which could cause the program to crash in some situations.

While this kind of bug can't be used by an attacker to take control over the execution of the program[4], it still undermines the stability of the software and could have serious implication in scenarios where even a simple crash could be problematic (e.g. think about the control system of an airplane).

Stack-based buffer overflow (*read_data*)
The function *read_data* contained a stack-based overflow vulnerability:

```
char name[NAME_LENGTH];
sscanf(command, "%*s %i %s", &age, name);
```

For instance if the length of the 3rd argument of the command was bigger than *NAME_LENGTH* then *sscanf* would have written beyond the bounds of the array *name*.

This kind of bug could have very serious security implication since it could be used by an attacker to take control over the execution flow of the application. This is most of the times done by overwriting the return address on top of the stack. Stack-based buffer could be, in the worst case, trivially exploitable if no further defenses are in place (stack canaries, stack shield, ASLR, etc..).
More complex attacks capable of defeating traditional defenses exists, for instance, if stack canaries are in use, one such attack could consist in overwriting the exception handler used in case of corrupted stack guard.

A simple fix for this bug could have been limiting the amount of characters read by *sscanf*. For instance by modifying the format string in *"%*s %i %19s"*, we would limit the number of characters read for the name to 19. Unfortunately such easy fix was not possible in our case since the use of *sscanf* was source of other unwanted behaviors (mostly due to its incapability to recognize some invalid inputs when parsing numeric values).

Format string bug (*handle_command*)
The function *handle_command* make use of an user-provided string as a format string:

```
fprintf(printFile, "No such command: ");
fprintf(printFile, command);
fprintf(printFile, "\n");
```

For instance, in case of an invalid command, the string *command* which is controlled by the user, is used as second argument for the function *fprintf* and therefore interpreted as a format string.
This bug could be used by an attacker to write or read arbitrary values in memory and therefore take control of the execution of the program or simply get access to sensitive informations. For instance an attacker could examine arbitrary addresses on the stack by using *"%m$x"* with an arbitrary offset m and write arbitrary values inserting the string *"%n"*.

[4]While this is true in our particular case, it should not be considered as a rule. For instance it could be possible to exploit a NULL dereference if the memory at the address 0 has been previously mapped by the program (despite this being unusual and involving the deactivation of some kernel protections) or in a kernel (since for reasons of performance kernels usually don't switch address space when passing in kernel mode)[3].

This bug is simply fixed by using the format string *"%s"* and passing *command* as the third argument:

```
fprintf(printFile, "No such command: ");
fprintf(printFile, "%s", command);
fprintf(printFile, "\n");
```

Invalid reads/writes on the heap (*read_command*)
The function *read_command* doesn't update properly the pointer *inputAt* after reallocating the memory:

```
input = realloc(input, sizeof(char) * inputMaxLength);
inputAt += incr - 1;
```

For instance, after reallocating the memory *input* could contain a different address than the previous one. On the other hand *inputAt* still references an offset from the previous value. This bug could be used by an attacker to overwrite some values on the heap and potentially take control over the execution flow of the program.
This bug is easily fixed by taking into account the new value of *input* while updating the pointer *inputAt*:

```
input = realloc(input, sizeof(char) * inputMaxLength); // Still a
    potential memory leak here
inputAt = input + inputMaxLength - INPUT_INCREMENT - 1;
```

C: Apply to an open source code base (bonus)

This section I discuss some of the bugs I found on some C/C++ code bases from github. Part 1. describes the content of the directory containing the files regarding this part of the assignment. Part 2. briefly illustrate the approach used in order to identify the bugs. In part 3. I describe the findings and propose a solution which in some cases resulted in a pull request to the original code base.

1. Files

For obvious reasons of space it was not possible to include in the report the complete source code of the faulty functions. The fact that some bugs are produced by the combined actions of multiple functions on the same object makes things even more difficult.
For this report I tried to cut out all the not bug-related parts in order to show only the interesting sections of the code.
Nonetheless, in order to give to the reader the opportunity to easily verify the bugs on the complete code, I decided to include a folder containing the integral files with the functions in object. The folder contains also the output of the tools that let me identify the bugs.

2. Approach

Similarly to the approach taken in the previous section, I tried to find some bugs using static analysis tools at first (*cppcheck*). This was easy since it didn't require me to build each application (which often implies installing the necessary

dependencies). Afterward I tried to execute the application using a dynamic analysis tools such as *valgrind* or *ASan*, providing all sort of bad inputs.

3. Results

The repository at `https://github.com/calccrypto/tar` contains a simple implementation of the tool *tar*. In particular, the project proposes a library containing a set of functions useful to deal with tar archives but it also contain the code for a ready-to-use tool which uses the function of the library and provides a similar interface to the one of the well known tool *tar*.

With the help of *valgrind*, after running the program multiple times with different inputs I was able to discover the following two bugs:

Use of uninitialized value
I was able to identify this bug after noticing *valgrind*'s message: "Conditional jump or move depends on uninitialised value(s)".

```
int tar_read(const int fd, struct tar_t ** archive, const char
    verbosity){
  ...
  struct tar_t ** tar = archive;
  ...
  for(count = 0; ; count++){
       *tar = malloc(sizeof(struct tar_t));
       if (update && (read_size(fd, (*tar) -> block, 512) != 512)){
            V_PRINT(stderr, "Error: Bad read. Stopping\n");
            tar_free(*tar);
            *tar = NULL;
            break;
       }
       ...
    /* Performs some checks and breaks out of the
         for loop if this was the last block        */
       ...

     tar = &((*tar) -> next);
  }
    return count;
}

void tar_free(struct tar_t * archive){
    while (archive){
       struct tar_t * next = archive -> next;
       free(archive);
       archive = next;
    }
}
```

Here the field *next* of the last archive of the linked list is never initialized (function *tar_read*) but it is still used in *tar_free*. Most of the times its value happens to be NULL so the program would behave correctly, (and this is probably why this bug was left unnoticed by the creators of this project) but if the program would allocate, write to and free some data on the heap before the allocation of the memory last archive, the field *next* could actually contain a value different

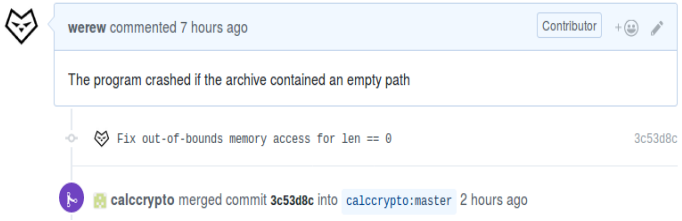
than NULL (maybe controlled by the attacker). In this case an attacker could probably exploit this bug using an approach not too different from the one used to exploit Use-After-Free vulnerabilities.
As final remark I would like to highlight another minor bug present in the previous code: the return value of *malloc* is used without any previous check.

Invalid memory access
This bug was particularly easy to discover since the program crashed if it was given some corrupted tar archives or just files that were not tar archives at all.

```
int extract_entry(const int fd, struct tar_t * entry, const char
    verbosity){
    V_PRINT(stdout, "%s\n", entry -> name);

    if ((entry -> type == REGULAR) || (entry -> type == NORMAL) ||
    (entry -> type == CONTIGUOUS)){
        // create intermediate directories
        size_t len = strlen(entry -> name);
        char * path = calloc(len + 1, sizeof(char));
        strncpy(path, entry -> name, len);

        // remove file from path
        while (--len && (path[len] != '/'));  // <---- BUG: What if
     len == 0 ?
        path[len] = '\0';    // if nothing was found, path is
    terminated

      ....
      ....
    }
    ....
}
```

On the above code it can be observed that if the *len* would be zero (in case of an empty name) the program would traverse all the memory looking for a '/' character. Therefore it will most of the time read or write to invalid memory raising an exception. Though this bug is hardly exploitable, it could be easily used by an attacker to crash the program (which could be an interesting DOS scenario in case of a remote server accepting tar archives to unpack).
This bug was easily fixed by checking the value of *len* before entering the loop. Figure 1 shows my pull request which has already been merged with the original repository.

Figure 1: Pull request



Memory leak
To conclude I would like to include one of the bugs found with *cppcheck*. I

have chosen to include this bug, among the other findings of *cppcheck*, because it shows a perfect example of a bug that could be very difficult to find using dynamic analysis tool (since some conditions should be met in order to trigger the unwanted behavior) but not for static analysis tools. This bug was found in the repository `https://github.com/clibs/commander` which is a library offering convenient functions to handle command line options.

```
static char **
normalize_args(int *argc, char **argv) {
  ...
  char **nargv = malloc(alloc * sizeof(char *));
  for (i = 0; argv[i]; ++i) {
  ...
    if (len > 2 && '-' == arg[0] && !strchr(arg + 1, '-')) {
      alloc += len - 2;
      nargv = realloc(nargv, alloc * sizeof(char *));
      ...
      continue;
    }
  }
  ...
}
```

The problem in question is easily spotted on the above code: there is no control over the return value of *realloc*. In particular, in case of failure of *realloc* the memory previously allocated and pointed by *nargv* would be permanently lost.

References

[1] ASAN, Google, `https://github.com/google/sanitizers`, c2017, last access: 2 December 2017.

[2] N. Nethercote, J. Seward, *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*

[3] nelhage , https://blogs.oracle.com/ksplice/much-ado-about-null\%3a-exploiting-a-kernel-null-dereference, c2010, last access: 2 December 2017.