The Bad Program

The subtitle

Shiva Prasad Gyawali, Shirinshoev Azamat

27 November 2024

1 Introduction

In this practical session, we will learn and apply our knowledge on threat finndings, their mitigations using compilation options, or directly modifying the code itself. Lastly, we will present our some recommendation on improving the process.

2 Options used for Legacy version

In this, we explored the various options that might have been used to compile our original door-locker binary. The compilation flags we will put into `LEGCFLAGS=` and the linker flags we will put `LEGLDFRLAGS=`.

2.1 Compilation options (LEGCFLAGS)

2.1.1 `-fno-stack-protector`

With the use of checksec tool, we saw that the provided `door-locker` binary doesn't have the canary. Thus, we assume that the flags `-fno-stack-protector` must have been used to disable stack protection. Stack protection is a security feature that helps prevent stack buffer overflow attacks. It works by inserting a stack canary value before the return address in a function's stack frame. During execution, the program checks this canary value before returning from a function. If the canary value has been altered (indicating a potential overflow), the program terminates with a security error. For security purpose, stack-protection must be enabled. Thus, either `-fstack-protection` or `-fstack-protection-all` options must be used.

2.1.2 `-D_FORTIFY_SOURCE=0`

`D_FORTIFY_SOURCE` option enables or disables additional buffer overflow checks for standard library functions. It checks the size of buffers and flags issues if they exceed their allocated memory. However, for security purpose, it's value must be 2.

2.1.3 "

2.2 Linker options (LEGLDFRLAGS)

2.2.1 `-m32`

We could check if the program was compiled as 32-bit or 64-bit by running **file door-locker**, which gives us ELF 32. It indicated the file type and architecture which in our case was ELF 32-bit (Linux). The `-m32` flag instructs the compiler and linker to generate a 32-bit binary, regardless of the host system's architecture (commonly 64-bit).

3 Identified threats and their mitigation

In this section, we will try to implement the mitigations we proposed in previous report.

3.1 Mitigations for Buffer Overflow

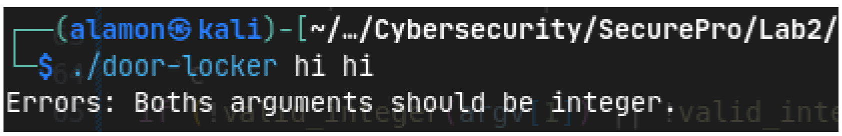
To address the identified vulnerabilities, we implemented several patches to the source code. The first patch involved replacing the scanf function with fgets, which allowed for safer input handling by limiting the number of characters read. The modified code included checks to ensure that the input did not exceed the buffer size, thus mitigating the risk of buffer overflow.

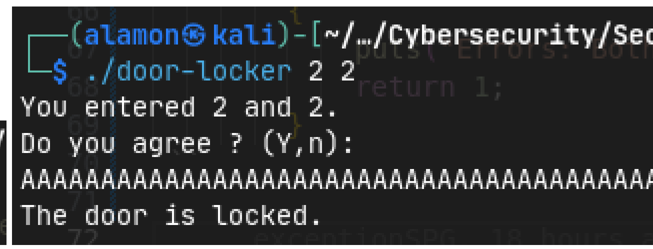
```
if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
  return -1;
}
```

Furthermore, we added input validation to ensure that only numeric character were processed by the strtol function, which we noticed in binary reverse engineering. However, when looking at the source code we found out that atoi is used, which is less secure. We created **valid_integer()** function to check each character in the input, thereby preventing invalid input from being treated as numeric.

```
if (!valid_integer(argv[1]) || !valid_integer(argv[2]))
        {
            puts("Errors: Boths arguments should be integer.");
            return 1;
        }
```

Output of the program is shown as below after implementing these changes:





3.2 Mitigations for visibility of Sensitive function

As we observed, `fnR()` was initialized but never used. However, there was a possibility that the attacker can utilize this function with the buffer-overflow, to gain the system access. Thus, we proposed hiding it or removing it from the executable binary. To mitigate this: 1. Compilation option 2. Changing `fnR()` code Instead of `system()` instruction, we are using `execv()`.

3.3 alternative of 'strtol'

3.4 Improved user feedback and error message

3.5 Recommendation for future processes