Assignment 3

secure programming 2016

Dimitri Diomaiuta - 2553339

Question 1:

The bug in this code is given by the use of the `gets()` function. This is a dangerous function that can cause a buffer overflow vulnerability that could be use by malicious user to exploit the program and gain full control over it. This is also reported by the man page relative to `gets()` under the BUGS section: "*Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.*"

As it is suggested in the `gets()` man page a possible fix is to use, instead of `gets()`, its bounded version, called `fgets()`. This function allows to bound the number of read characters from standard input in order to prevent buffer overflow vulnerability. The fixed version of the code can be find in the problem1_solved.c file under the same directory where this README.pdf file is present. Since `fgets()` is called just one time there is no need to flush the input that has not been read. This will be done by the operating system once the program is terminated.

Question 2:

All the three properties composing the CIA triad are broken:

1. **Confidentiality** is violated since once a malicious user takes control of the program, exploiting a buffer overflow vulnerability, can gain access to other users' private information.
2. **Integrity** is violated since once the buffer overflow vulnerability is exploited the attacker can change the data the program deals with making it inconsistent and not accurate anymore.
3. **Availability** is violated since when an hacker breaks into the system, by exploiting a buffer overflow vulnerability, he can gain full control of the program. He can then block the service and deny it to authorized users, making the program follow his willings.

Question 3:

The problem in this code is that there is no integer overflow prevention. The sum of two integers can lead to an integer overflow, consisting in the sum being bigger than the max value that a type of integer can represent. In this case two unsigned short numbers are added. The result is then used to allocate that quantity of memory. If an integer overflow occur than the program allocates less memory than what it needs, resulting in a possibly buggy execution of the program. In order to check this a conditional statement should be added before doing the sum operation:

```
if (a > USHRT_MAX - b || b > USHRT_MAX -a) {
    abort();
  }
```

This conditional statements aborts the program in the case the addition of the two unsigned shorts will lead to a value bigger than $2^{16}-1$, that is 65535 (since unsigned short are represented by 16 bits). There is no need to check whether the unsigned short are less than 0 since unsigned offer type safety for negative values. In the case the addition of the two numbers do not cause any integer overflow than the execution of the program can continue, allocating the right quantity of memory.

The fixed code of this program can be found in the problem3_solved.c file under the same directory of this README.pdf file.

Question 4:

The program has two bugs:

1. The first one is the use of the `gets()` (line 9)function. As seen in the answer to the first question this function is not bounded hence do not provided control in preventing buffer overflows vulnerability.
2. The second one is the `strncpy()` (line 21) function. This function is bounded and safer to use than its unbounded sister `strcpy()`. The problem here is given by the last parameter passed to the function, that is the number of bytes to copy from the source to the destination buffer. In this case the size of the destination buffer is given as last paramter: `strncpy(buf, name, sizeof(buf));` (line 21). This results in buf not being null terminated. This error is an index error known as the off by one error. If we analyze the `strncpy()` function from its man page we can see that the null character is never appended at the end of the destination buffer since the second loop is never entered:

   ```
   char * strncpy(char *dest, const char *src, size_t n) {
       size_t i;
       for (i = 0; i < n && src[i] != '\0'; i++)
          dest[i] = src[i];
       for ( ; i < n; i++)
           dest[i] = '\0';
       return dest;
   }
   ```

   In the case the program is compiled with the `-fno-stack-protector` flag this off-by-one can cause the program to have a buggy execution since the buf (the destination character array variable of the `strncpy()` function) will be printed in the next line but is not null terminated. This can make `printf()` read more than it should do and hence access other elements of the stack.

A) In the case the user provides a large input the return address will be overwritten and a buffer overflow will occur. In order to know how many characters a user should input to overflow the return address the program should be compile with gcc and the -static flag on (in order not to have dynamic libraries and to obtain a blob containing all the function calls compiled within the program) and then disassemble it with the gdb program. We call gdb with: `gdb offby1.c` and then we type `disassemble get_name`. This will result in obtaining the assembler calls made by the get_name function (the one containing the overflow bug):

```
Dump of assembler code for function get_name:
   0x0000000000400fde <+0>:      push   %rbp
   0x0000000000400fdf <+1>:      mov    %rsp,%rbp
   0x0000000000400fe2 <+4>:      sub    $0x30,%rsp
   0x0000000000400fe6 <+8>:      mov    %rdi,-0x28(%rbp)
   0x0000000000400fea <+12>:     mov    %rsi,-0x30(%rbp)
   0x0000000000400fee <+16>:     mov    -0x30(%rbp),%rax
   0x0000000000400ff2 <+20>:     mov    %rax,%rsi
   0x0000000000400ff5 <+23>:     mov    $0x48f848,%edi
   0x0000000000400ffa <+28>:     mov    $0x0,%eax
   0x0000000000400fff <+33>:     callq  0x407500 <printf>
   0x0000000000401004 <+38>:     lea    -0x20(%rbp),%rax
   0x0000000000401008 <+42>:     mov    %rax,%rdi
   0x000000000040100b <+45>:     callq  0x407ca0 <gets>
   0x0000000000401010 <+50>:     lea    -0x20(%rbp),%rcx
   0x0000000000401014 <+54>:     mov    -0x28(%rbp),%rax
   0x0000000000401018 <+58>:     mov    $0x14,%edx
   0x000000000040101d <+63>:     mov    %rcx,%rsi
   0x0000000000401020 <+66>:     mov    %rax,%rdi
   0x0000000000401023 <+69>:     callq  0x400340
   0x0000000000401028 <+74>:     leaveq
   0x0000000000401029 <+75>:     retq
End of assembler dump.
```

The `printf()` call is immediately before the `gets()` call. We are, as gdb suggests, 33 bytes from the return address, hence an input of 33+1 characters will overflow the return address and arrest the execution of the program because of a segmentation fault:

```
./offby1
Enter your name:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

This can be used by an attacker to gain control of the program and to make the return address point to what he prefers. This can lead to violation of data confidentiality, by reading for example to the secret array assigned value, data integrity, by changing the value of the secret array, and availability. The last one can be assessed if the hacker spawns a shell with the `execve()` function call (as it is explained in the “smashing the stack for fun and profit” article). Once gained control of a shell the attacker can deny the service and do so much more.

B) The fixed version of the offby1.c program can be found in the same directory of this README.pdf file under the name problem4_solved.c. The first bug is fixed by changing the `gets()` function with the safer `fgets()` one. Next a flush while loop is added in order to discard the rest of the characters that the user has input (in order not to affect the second `fgets()` during the second call to `get_name()` function). This is achieved by the following lines of code:

```
while ((ch = getchar()) != '\n' && ch != EOF) {

}
```

The second bug (being buf a not null terminated array of characters) is solved by passing to `strncpy()` function a n-1 bytes as the number of bytes to be read from the source array:

```
strncpy(buf, name, sizeof(buf)-1);
```