EECS 444 Homework 1 Part 1

Yida Liu

September 22, 2019

1 Problem 1

In-class exercise.

2 Problem 2

2.1 What are the ouputs for the corresponding inputs?

Input 1

The output is `String after concatenation: |dest+src|`

Input 2

The output is

```
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
```

2.2 Code Fix

For the above code, there are several vulnerabilities. For example, the usage of unsafe `gets` and `strcat`, which might raise buffer overflow and heap corruption. We fixed the problem by adding the following changes:

1. Increase the buffer size of `dest`

   The main purpose of the program is to concatenate the two user input string. Therefore, it is unreasonable to have the two buffer the same size. Therefore, if we want to limit the user input to 30 characters (per line), the destination buffer should be twice the size of that, 60.

2. Replace `gets` with safer `fgets`

   `fgets` limits the maximum number of characters to be read from the input stream, which is considered a safer function to use to read from `stdin`.

3. Flush the `stdin` for the following `fgets` on the second buffer.

   The problem is that if there is more to be read in `stdin` after the first `fgets`, the second `fgets` will directly read the rest from `stdin`, instead of blocking until the user enters the second input. Therefore, we manually flush the `stdin` by continuous reading characters until an `EOF` or `'\n'` is reached.

4. Check the size of concatenation from `src` to `dest`

   We check the remaning size of buffer `dest` to how many extra character can be concatenated and used `strncat` for concat if there is at least 1 empty space. At the same time, we also compare the string length of `src` and the remaining length in `dest` and pick the smaller for the parameters to avoid possible out-of-bound problem.

5. Remove trailing white space from `fgets`

The following code is the complete program after applying the aforementioned updates to the give code.

```c
#include <stdio.h>
#include <string.h>
// 4. Define min
#ifndef min
    #define min(a,b) ((a) < (b) ? (a) : (b))
#endif

int main() {
    // 1. Change buffer size
    // input length on each line
    int IN_LEN = 30;
    // total input length, which is the size of dest
    int TOT_IN_LEN = IN_LEN * 2;

    char dest[TOT_IN_LEN], src[IN_LEN];

    // 2. Replace gets with fgets
    fgets(src, IN_LEN, stdin);

    // 5. Remove trailing newline if there is any
    char *nl;
    if ((nl=strchr(src, '\n')) != NULL) *nl = '\0';

    // 3. Flush stdin after first read
    if (!strchr(src, '\n')){
        int c;
        while ((c=getchar()) != '\n' && c != EOF);
    }

    // 2. Replace gets with fgets
    fgets(dest, IN_LEN, stdin);

    // 5. Remove trailing newline if there is any
    if ((nl=strchr(dest, '\n')) != NULL) *nl = '\0';

    // 4. Check size and use strncat
    int rem_char_in_dest = sizeof(dest) - strlen(dest) - 1;
    if ( rem_char_in_dest > 0 ) {
        strncat(dest, src, min(rem_char_in_dest, strlen(src));
    }

    printf("String after concatenation: |%s|", dest);
    return(0);
}
```

2.3 Abuse Cases

We provide a list of illegal inputs (and / or their combinations) for `src` and `dest` and explain how our revised program defend them.

- Long input (length $\geq$ `IN_LEN`)

  `fgets` will only capture the first `IN_LEN` number of characters; Additional characters in `stdin` is removed as indicated in Section 2.2 Fix. 3; Section 2.2 Fix. 4 makes sure that `strncat` will not have out-of-bound accesses of string elements.

- Empty input (length = 0, newline character only)

  Section 2.2 Fix. 5 removes the trailing newline when obtaining string from `fgets`.

3 Problem 3: Integer Overflow

The defect for this function is that it assumes the positivity of `len1` and `len2` without actually validating it or using data types that enforces positive input. The following case will break the code: when `len1 = 2147483647` and `len2 = -2147482625 = 1022 - len1`

1. the sanity check at line 1 will fail to detect the sizes
2. the buffer initializer will get `pBuf` of size 1023
3. the `memcpy` from s1 to `pBuf` will cause index out of bound
4. the `memcpy` from s2 to `pBuf + len1` will copy a large chunk of data that might or might not belong to s2 to an out-of-bound location

4 Problem 4: Integer Overflow 2

As from table 1, we see that the length of each datatype are different. In line 2, there is an downsizing from signed long to usngiend short; In line 8, function `memcpy` actually takes in an unsigned integral type for size, instead of signed. For this two part, overflow could happen that influence the behavior of the code. Suppose `cbBuf = 2147483648 = INT_MAX + 1`,

1. `cbCalculatedBufSize` will become the lower word of `cbBuf`
2. `cbBuf` will be interpreted as unsigned long when passing in `memcpy`.

Numeric Type	Bytes	Range
unsigned short	2	$[0, 65535]$
int	4	$[-2147483648, 2147483647]$
long	8	$[-9223372036854775808, 9223372036854775807]$

Table 1: Range of Common Numeric Types