CS 61 Lecture 6

Memory Errors

I. Stack buffer overflow error

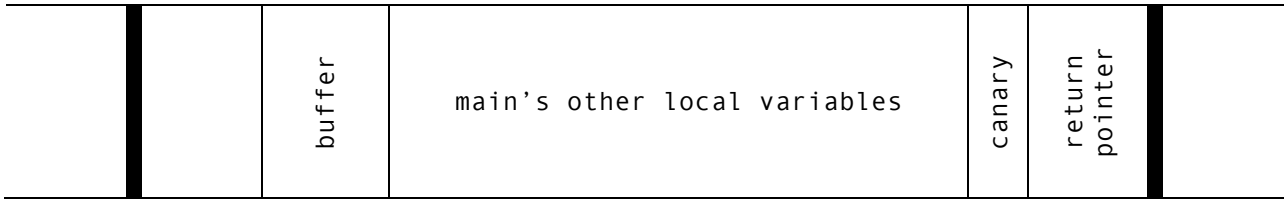
- Consider the following program, `stacksmashf.c`, which may be found in the `l06` directory:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 int read_line(char *buffer) {
 6     if (gets(buffer))
 7         return 1;
 8     else
 9         return 0;
10 }
11
12 int main() {
13     char buffer[100];
14     if (read_line(buffer))
15         printf("Read %d character(s)\n", strlen(buffer));
16     else
17         printf("Read nothing\n");
18 }
```

- As written, it contains a "stack buffer overflow error"
  - Why "stack"?
    - Bcause it's dealing with local variables (see line 13)
    - [N.B. the stack is located towards the end of memory, about ¾ of the way down]
  - What is a "buffer"?
    - A variable defined to hold user input or program output formatted for external use
  - What is an "overflow"?
    - When the program runs over the bounds of a buffer
- If the program runs over by too many bytes, it will result in a segmentation fault
  - The segmentation fault happens at different times/places in the code depending on how much writing `gets` does
- But when run without the `.unsafe` suffix, the processors knows that the program was trying to "stack smash" (i.e. write beyond the bounds of the buffer)
  - How does it know…

II. Canary in a coal mine

- Stack canaries, or sentinels, are used to detect a stack buffer overflow before execution of malicious code can occur
- Where to put the canary?
  - Option A: right after each buffer
  - Option B: right before the return address
    - [N.B. in a 32-bit architecture, the return address is always the top 4 bytes of the stack frame (i.e. those furthest to the "right")]
- Pros and cons of options A and B
  - Pro of A: catch even the smallest of overflows
  - Con of B: won't catch all overflows, some of which may still cause the program to crash
  - Con of A: the program is slower, especially when multiple buffers exist
  - Pro of B: faster and takes up less space



A close up view of a stack frame for stacksmash.c

- Which option does `gcc` choose? Option B
  - On function entry, `gcc` chooses a canary value and stores it in the stack frame very close to the return address
    - [N.B. Since most buffer overflows overwrite memory from lower to higher memory addresses, in order to overwrite the return pointer and take control of the process, the canary value must also be overwritten]
  - On function exit, check the canary value and if it has been changed, stop executing the program
- What canary value does `gcc` choose?
  - A random number
    - [N.B. a typical canary on a 32-bit machine is 4 bytes long]
  - A hacker would be able to deduce a constant value

III. Heap buffer overflow error

- Consider the following program, `heapsmashf.c`, which may be found in the `l06` directory:

```
1 #include <stdlib.h>
2 #include <string.h>
```

```
 3
 4 int read_line(char *buffer) {
 5     if (gets(buffer))
 6         return 1;
 7     else
 8         return 0;
 9 }
10
11 int main() {
12     char *buffer = malloc(100);
13     if (read_line(buffer))
14         printf("Read %d character(s)\n", strlen(buffer));
15     else
16         printf("Read nothing\n");
17 }
```

- Unlike in the previous program, the buffer is now stored on the heap
    - On the heap, the buffer is not located near the return pointer of the function
    - [N.B. the heap is located towards the start of memory]
- There are two memory errors in the above code: 1) heap buffer overflow error and 2) a memory leak
    - In general, memory leaks are less desirable because they reduce the utilization of the computer's memory
    - In the case of this program, the memory leak is not that serious
- When `free(buffer);` is added to the code, the compiler adds a canary to the end of the buffer in memory
    - With this call, the program will have a segmentation fault more often

IV. Security stubs

- Consider the following program, `stacksmash.c`, which may be found in the `l06` directory:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 int main() {
 6     char buffer[100];
 7     if (gets(buffer))
 8         printf("Read %d character(s)\n", strlen(buffer));
 9     else
10         printf("Read nothing\n");
11 }
```

- When this program is run, a "buffer overflow" is reported rather than "stack smashing detected"
    - In addition, the `printf` call on line 8 does not execute
- Why the difference between this program and `stacksmashf.c`?
    - In this case, the call to `gets` is in the same function as the buffer
- In response, the compiler implements a security stub
    - Security stub = a version of a function with additional error checking
    - The compiler know how big the buffer is, and so it doesn't let `gets` write to a larger portion of memory
    - The compiler swaps the security stub function that checks for buffer overflows in automatically for `gets`
    - In `stacksmashf.c`, this swap does not happen

V. Integer overflow error

- Consider the following function from `indexsmash.c`, which may be found in the `l06` directory:

```
 6 void read_array_element(int *array, int size,
 7                         char *index_arg, char *value_arg) {
 8     long index = strtol(index_arg, NULL, 0);
 9     long value = strtol(value_arg, NULL, 0);
10     assert(index < size);
11     array[index] = value;
12 }
```

- Running this function doesn't release nasal demons if you try to go beyond the bounds of the array
    - Why? See the security stub in line 10
- However, if you enter a negative integer for the array index, you will cause an error by entering the stack frame to the "left" of yours
    - You could even reset its return pointer!
    - There's not a security stub to check for this input
    - But one may be added via another assert call: `assert(index >= 0);`

VI. Growable Buffers

- Consider the following function from `linebuffer.c`, which may be found in the `l06` directory:

```
17 size_t linebuffer_gets(linebuffer *lb, FILE *stream) {
18     size_t length = 0;
19     int c;
20     while ((c = fgetc(stream)) != EOF) {
```

```
21          lb->buffer[length] = c;
22          ++length;
23          if (c == '\n')
24              break;
25      }
26      lb->buffer[length] = 0;
27      return length;
28  }
```

- A growable buffer utilizes dynamically allocated memory
- This version of `linebuffer_gets` will produce a segmentation fault if too many characters are read from `stdin`
- The best option for turning this static sized buffer into a growable buffer is to utilize a doubling policy
    - When the length of what we are writing into memory is bigger than or equal to the capacity of the buffer, call `realloc` for a buffer of twice the capacity
    - When writing this code, ensure that the initial capacity of the buffer is 1, not 0
- Why double?
    - Doubling will run in O(n) rather than $O(n^2)$
- For an example of a growable buffer we've all likely used, see below for a copy of the CS50 library function, `GetString`

```
string GetString(void)
{
    // growable buffer for chars
    string buffer = NULL;

    // capacity of buffer
    unsigned int capacity = 0;

    // number of chars actually in buffer
    unsigned int n = 0;

    // character read or EOF
    int c;

    // iteratively get chars from standard input
    while ((c = fgetc(stdin)) != '\n' && c != EOF)
    {
        // grow buffer if necessary
        if (n + 1 > capacity)
        {
            // determine new capacity: start at 32 then double
            if (capacity == 0)
                capacity = 32;
            else if (capacity <= (UINT_MAX / 2))
```

```
                capacity *= 2;
            else
            {
                free(buffer);
                return NULL;
            }

            // extend buffer's capacity
            string temp = realloc(buffer, capacity *
sizeof(char));
            if (temp == NULL)
            {
                free(buffer);
                return NULL;
            }
            buffer = temp;
        }

        // append current character to buffer
        buffer[n++] = c;
    }

    // return NULL if user provided no input
    if (n == 0 && c == EOF)
        return NULL;

    // minimize buffer
    string minimal = malloc((n + 1) * sizeof(char));
    strncpy(minimal, buffer, n);
    free(buffer);

    // terminate string
    minimal[n] = '\0';

    // return string
    return minimal;
}
```