ECE 264: Advanced C Programming

Programming Assignment 2 - Program Stack, Due 6/23/19

Many students think programs are correct if they can produce expected outputs. This is far from the truth. Programs may have hidden mistakes that can cause serious problems when certain conditions are met. One common mistake is accessing (reading or writing) invalid memory locations.

When the C language was invented, computers were very expensive and few people were able to use computers. Those people had to take multiple courses before they could use computers. At that time, one of the highest priorities was to make computer programs run fast. As a result, there were many security holes in early computers. One of the most exploited holes is called "buffer overflow".

1 Learning Goals

You will learn

- Restrictions of Array Indexes
- Use gdb to understand the callstack
- Structure of the call stack (also called stack memory)
- Express values in hexidecimal system
- Calculate the distance between an array's starting address to the place where a return location is stored
- Simulate buffer overflow attack by modifying an array's elements using invalid indexes

2 Background

2.1 Array Indexes

If a C array has n elements, valid indexes are `0` to `n - 1` (including `0` and `n - 1`). The first element has index `0`, not `0`. This is always true and never changes.

This assignment includes several examples showing how invalid indexes could affect programs. In this first example, `wrongindex1.c`, wrong indexes `5, 6, 7, -1, and -2` were used. The value of `y` was changed by index `-1`; the value of `x` was changed by index `7`. If the program never checks the value of `x` or `y`, the program may appear correct.

In the second example, `wrongindex2.c`, even though `x` was not passed to the function `f1`, the value of `x` is changed.

2.2 GDB

`gdb` is a debugger that allows you to interactively check the flow of a program as well as the values of variables. `gdb` allows you to single step a program or set conditional breakpoints. You can think of `gdb` as X-ray + CT + ultrasonic for inspecting a program.

2.3 Stack Memory

A computer's memory is organized as address-value pairs. A computer's memory is divided into two parts: user and kernel. The user memory is used by application programs. The kernel memory is used by the operating system. The user memory is further divided into three parts: program memory, stack memory, and heap memory.

The program memory stores the running program. The stack memory stores the data used for function calls. Neither the program memory nor the stack memory can be directly controlled by a user program. The heap memory, in contrast, is under direct control of user programs. A user program may allocate (`malloc` in C) or release (`free` in C) memory.

The stack memory follow a strict rule of "last in, first out". What is added (called "push") to the stack last is removed (called "pop") from the stack first. This rule is always followed and there is no exception in any circumstance. By convention, the place where new information is pushed or popped is called the "top" (in contrast to the bottom) of stack.

Whenever a function is called, a (one, only one, exactly one) "frame" belonging to the called function is pushed to the top of the stack memory. When the function finishes, the top (one, only one, exactly one) frame is popped from the stack memory. This rule is always followed and there is no exception in any circumstance.

A frame stores at least one piece of information: the location where the program continues after the called function finishes. This is called the "return location" and it refers to an address in the program memory. A frame may store additional information for the value address, argument(s), and local variable(s).

2.4 Hexidecimal Number System

Decimal numbers (0, 1, 2, ..., 9), which are base 10, are widely used in daily life but computers understand only binary numbers (i.e., 0 or 1). Binary numbers can be quite long, so another common representation is hexadecimal numbers (0, 1, 2, ..., 9, A, B, C, D, E, F), which are base 16. A hexadecimal number starts with 0x to indicate that it is hexadecimal. "Hexadecimal" is often called "hex" for short.

Hexadecimal numbers are a convenient representation because a group of four binary digits can be represented by one hex digit. Some examples are:

- 0000 = 0 in hex (0 in decimal)
- 1010 = A in hex (10 in decimal)
- 1111 = F in hex (15 in decimal)
- 1010 0111 = A7 in hex (= 10 * 16 + 7 = 167 in decimal)

2.5 Find Return Location

To correctly determine the return location in a frame, you need to understand the structure of the stack memory. This assignment uses a simpler method: by matching a function's addresses with the values stored in the stack memory.

Consider the program `wrongindex3.c`. Use `gcc` to create the executable. In PA01, we told you to always run `gcc` as follows:

```
gcc -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror
```

However, if you try to compile `wrongindex3.c`:

```
bash-4.1$ gcc -std=c99 -g -Wall -Wshadow -pedantic -Wvla -Werror wrongindex3.c -o wrongindex3
```

You will get an error. This is because `-Werror` forces `gcc` to treat all warnings as errors. In this assignment, we don't want that. If your `gcc` is aliased, *for this assignment only* unalias it:

```
bash-4.1$ unalias gcc
```

Now when you compile `wrongindex3.c`:

```
bash-4.1$ gcc wrongindex3.c -o wrongindex3 -g
```

you will see several warning messages that you can ignore.

This is the only assignment which allows warning messages.

Start `gdb`:

```
bash-4.1$ gdb wrongindex3
```

Please notice that the input to `gdb` is the executable, not a source file (i.e., no `.c`)

You can see the prompt is changed to (`gdb`)

Type

```
(gdb) disassemble/m main
```

You can see the following output

```
29  {
   0x0000000000400577 <+0>: push   %rbp
   0x0000000000400578 <+1>: mov    %rsp,%rbp
   0x000000000040057b <+4>: sub    $0x20,%rsp
   0x000000000040057f <+8>: mov    %edi,-0x14(%rbp)
   0x0000000000400582 <+11>:    mov    %rsi,-0x20(%rbp)

30      int main_top = 0xEEEEEEEE; // used as references
   0x0000000000400586 <+15>:    movl   $0xeeeeeeee,-0x8(%rbp)

31      f1();
   0x000000000040058d <+22>:    callq  0x4004cb <f1>

32      int main_btm = 0xFFFFFFFF; // used as references
   0x0000000000400592 <+27>:    movl   $0xffffffff,-0x4(%rbp)

33      return EXIT_SUCCESS;
   0x0000000000400599 <+34>:    mov    $0x0,%eax

34  }
   0x000000000040059e <+39>:    leaveq
   0x000000000040059f <+40>:    retq
```

The hexadecimal numbers on the left (e.g., `0x0000000000400689`) represent the addresses in memory where the assembly instructions on the right (`push`, `sub`, etc.) Please be aware that the specific addresses may be different in your program.

Next, type

```
(gdb) disass/m f1
```

This is the first several lines of the output

```
11  {
   0x00000000004004cb <+0>: push   %rbp
   0x00000000004004cc <+1>: mov    %rsp,%rbp
   0x00000000004004cf <+4>: sub    $0x20,%rsp
```

```
12      int f1_top = 0xAAAAAAAA; // used as references
   0x00000000004004d3 <+8>: movl   $0xaaaaaaaa,-0x8(%rbp)
```

Do you notice that **`0x4004cb`** appears again? It appears earlier in main:

```
31      f1();
   0x000000000040058d <+22>:    callq  0x4004cb <f1>
```

It appears again as the very beginning of `f1`

```
   0x00000000004004cb <+0>: push   %rbp
```

What does this mean? This is the address of the function `f1`. The line in main is calling `f1`, and it uses the address to tell where to jump to.

Similarly, it is possible to find the beginning of `f2`:

Type

```
(gdb) disass/m f2
```

This is the first few lines of the output

```
21  {
   0x000000000040051d <+0>: push   %rbp
   0x000000000040051e <+1>: mov    %rsp,%rbp
   0x0000000000400521 <+4>: sub    $0x30,%rsp

22      int f2_top = 0xCCCCCCCC; // used as references
   0x0000000000400525 <+8>: movl   $0xcccccccc,-0x8(%rbp)
```

The address of `f2` is `0x000000000040051d`.

Set a breakpoint at `f1`:

```
(gdb) b f1
```

Start the program:

```
(gdb) r
```

The program stops at the beginning of `f1`.

The command `x` in `gdb` displays the memory content. It can be followed by the amount of memory to be shown. For example,

`x/80bx` shows 80 bytes of memory in hexadecimal. In x86 (i.e., Intel) processors,`$rsp` is the stack pointer. The stack pointer is the very top of the stack memory.

Note that by convention, the "top" of the stack has the lowest address. When we add data to the stack, the stack pointer moves to a lower address. That is why `f2` has the instruction `sub $0x30,%rsp`: it's adding 48 bytes to the stack by subtracting 48 from the address stored in `$rsp`. (Don't forget your hex numbers! 0x30 in hex is 48)

If you type

```
(gdb) x/80bx $rsp
```

You will see:

```
0x7fffffffe3b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe3b8: 0xa3    0x03    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe3c0: 0xf8    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe3c8: 0xf5    0x05    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe3d0: 0x00    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
```

```
0x7fffffffe3d8: 0x92 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe3e0: 0xe8 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3e8: 0xe0 0x03 0x40 0x00 0x01 0x00 0x00 0x00
0x7fffffffe3f0: 0xe0 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3f8: 0xee 0xee 0xee 0xee 0x00 0x00 0x00 0x00
```

This shows 80 bytes of memory, starting from the stack pointer. Because the stack grows "down," this shows the last 80 bytes of data that have been added to the stack. The hexadecimal numbers on the left represent memory addresses (note that they are much larger numbers than the addresses of the program! This is because the addresses of the program are in program memory, which is a different portion of memory than the stack memory, which uses much higher addresses.)

If you pay careful attention, you may have noticed the value `0x0000000000400592` appears starting at address `0x7fffffffe3d8`:

```
0x7fffffffe3d8:  0x92 0x05 0x40 0x00 0x00 0x00 0x00 0x00
```

`0x0000000000400592` is the return location for function call `f1`.

```
32      int main_btm = 0xFFFFFFFF; // used as references
   0x0000000000400592 <+27>:    movl   \$0xffffffff,-0x4(%rbp)
```

The byte order appears reversed because Intel processors store numbers in "Little Endian" format. Rather than the least significant byte (0x92) being stored in the largest address (which is called "Big Endian"), the least significant byte is stored in the smallest address. So when you write out the bytes the "normal" way (starting at the smallest address), the bytes look backwards. The return location has 8 bytes because this example is created on a machine with 64-bit addresses.

Use the next (or `n`) command to go to the next line in `f1`:

```
(gdb) n
```

3 times. Now, type

```
(gdb) x/80bx $rsp
```

again and the output is

```
0x7fffffffe3b0: 0x09 0x08 0x07 0x06 0x05 0x04 0x03 0x02
0x7fffffffe3b8: 0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe3c0: 0xf8 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3c8: 0xaa 0xaa 0xaa 0xaa 0xbb 0xbb 0xbb 0xbb
0x7fffffffe3d0: 0x00 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3d8: 0x92 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe3e0: 0xe8 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3e8: 0xe0 0x03 0x40 0x00 0x01 0x00 0x00 0x00
0x7fffffffe3f0: 0xe0 0xe4 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe3f8: 0xee 0xee 0xee 0xee 0x00 0x00 0x00 0x00
```

Did you notice that the values starting from `0x7fffffffe3b0` appear to be the values of the `name` elements?

```
0x7fffffffe3b0: 0x09 0x08 0x07 0x06 0x05 0x04 0x03 0x02
0x7fffffffe3b8: 0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00
```

You can verify that by typing

```
(gdb) print &name
```

Which will print the address where the variable `name` is stored in memory:

```
$1 = (char (*)[10]) 0x7fffffffe3b0
```

2.6 Buffer Overflow Attack

This program modifies the values of `name[40]` and `name[41]`. The indexes are invalid and they happen to be the memory where the return location is stored. If you count up 40 bytes (5 rows of 8 bytes) from the location where `name` starts (`0x7fffffffe3b0`) you get to `0x7fffffffe3d8` which is the location of the return address we found from above.

`name[40]`, in C, just means "40 `char`s from the beginning of `name`. (It's 40 `char`s because name is an array of `char`s. If `name` were an array of `int`s, `name[40]` would mean "40 `int`s from the beginning of `name`.) A `char` takes up 1 byte, so `name[40]` means 40 bytes from the beginning of `name`. If we write to those bytes, we will overwrite whatever data is stored in location `0x7fffffffe3d8`! We will change the return address, and when we get to the end of `f1`, we will use the wrong address to return to!

What values should the program modify to? The value should be the beginning of `f2` (from `disass/m f2`).

```
name[40] = 0x1d;
name[41] = 0x05;
```

(The exact location of `f2` may be different for you, so the values of `name[40]` and `name[41]` might need to be changed to make this example work.)

If we change those locations, that will change this portion of memory:

```
0x7fffffffe3d8:  0x92 0x05 0x40 0x00 0x00 0x00 0x00 0x00
```

to:

```
0x7fffffffe3d8:  0x1d 0x05 0x40 0x00 0x00 0x00 0x00 0x00
```

Now, if you continue the program:

```
(gdb) c
```

The program outputs

```
REALLY BAD IF YOU SEE THIS
```

Where does this message come from? It is from `f2`. However, the program never calls `f2`! How can it be possible to execute code in `f2`? Because your program uses bad indexes (40 and 41) to modify the return location.

This program will then cause a "segmentation fault" because the operating system detects something wrong and stops the program. Nevertheless, you can say the damage has been done.

This is the only assignment where "segmentation fault" is tolerated for the purpose of explaining the potential damage of buffer overflow.

If you change the indexes to something else (still invalid, for example, 20 and 21), it appears that nothing bad happens. However, this is not true. The wrong indexes are still wrong.

Many students believe that if nothing bad appears, their programs are correct. This is far from the truth. This example shows that when mistakes happen to be at specific places, something really bad can happen.

3 What do you need to submit?

You need to modify `pa02.c` so that the program prints

```
REALLY BAD IF YOU SEE THIS
```

but the program must not call `f2` directly.

What you need to do is to determine the indexes in name and the values so that the program modifies the return location to the beginning of `f2`. Use `gdb` to do this using a similar technique as we saw in the background section.

4 Acknowledgments

Some materials in this assignment came from an assignment https://engineering.purdue.edu/ece264/16au/hw/HW15 used in Fall 2016 created by Professor Alex Quinn