Buffer Explanation

1 Understanding Stack

Before we get into more assignment details, it is important that you understand how stack frames get set up in the x86 architecture when functions are called. **Below is a brief overview, but if you would like to get more information, we highly encourage you to review lecture materials and go to TA hours.**

First of all, what is the stack? It is a region of memory that is used to store information about function calls. The stack consists of stack frames, where a stack frame is some amount of memory allocated for a function call, used to keep the information pertaining to that particular function invocation, such as local variables. There is one stack frame per each function call. Every time a function is called, we allocate a new stack frame, and every time a function exits, the stack frame for that call becomes available memory to use again.

Suppose we have nested invocations, like function A calling function B, which calls function C. Then, we first create a stack frame for A. Once A calls B, we make a stack frame for B below A's stack frame. And lastly, we make a stack frame for C below that of B. Once C exits, we pop off its stack frame, and go back to B. Once B exits, we pop off the stack frame for B, and lastly, we pop off A's frame. As you can see, the stack is managed in the LIFO order (last in, first out).

Let's take a closer look at how a stack frame is managed. The base register, *rbp*, points to the beginning of the current stack frame, and the stack pointer, *rsp*, points to the end of the current stack frame. The stack can grow by decreasing the value of *rsp* (remember that the stack starts at a high memory address and grows toward low memory addresses).

Note: in x86-64, we usually do not use the base register (rbp). However, because we are compiling our programs with low optimization (with the -O0 flag) to make the assignment easier to work through, *rbp* is still used in this assignment and we are not relying on *rsp* as a means of keeping track of things on the stack.

Here are two example functions in C:

```
int callee(int a, int b) {
      int newA = a + 2;
```

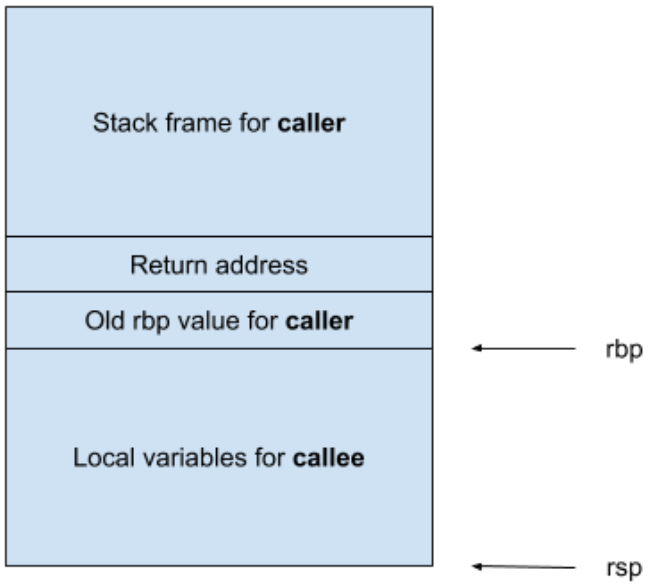
```
	int newB = b + 3;
	return newA + newB;
}

int caller() {
	int a = 2;
	int b = 3;
	int c = callee(a, b);
	printf(“%d”, c);
}
```

When the function **`caller`** calls the function **`callee`**, a new stack frame is prepared for the callee. First, it pushes a return address onto the stack. A return address is the address of the next instruction after coming back from **`callee`**, so in this case it would be the address of “`printf(“%d”, c);`”.

The next step is pushing the value of *rbp*. Currently *rbp* stores the beginning address of **`caller`**’s stack frame. The reason why we do this step is because *rbp* now needs to store the beginning address of **`callee`**’s stack frame, but once **`callee`** is done, we will need to return to **`caller`**‘s stack frame and restore the beginning address of **`caller`**‘s stack frame in *rbp*. So we temporarily push this value on the stack so we can retrieve it later.

The stack now looks like the diagram below:



And lastly, in x86-64, arguments are passed through registers. Specifically, *rdi* is used to store the first argument (for **`callee`**, this would be the argument **`a`**), *rsi* for the second argument (the argument **`b`**), *rdx* for the third argument, so on and so forth. Return values are held in *rax*. Check out the x86-64 Guide, specifically section 4.3, if you want to learn more about registers.

Now, the stack grows as necessary to store local variables for **`callee`** (in this case, **`newA`** and **`newB`**). We perform the computation, and once we hit the return statement, the stack shrinks (*rsp* is incremented), and it pops the old *rbp* value we pushed onto the stack previously (so now rbp points to the beginning of **`caller`**'s stack frame). Lastly it pops off the return address (into *rip*), and control of our program returns to the next instruction to execute in **`caller`**. *rip* is a register that keeps track of the next instruction to be executed.

2 Cookies

Phases of this project will require a slightly different solution from each student. To make each submission unique, we are using the **`makecookie`** program which will generate a *cookie* or *hash*.

A *cookie* or *hash* is a string of eight hexadecimal digits generated when you run the **`makecookie`** program. Each time you run **`makecookie`**, you will receive a different cookie. For example:

```
./makecookie
0x5d968887
```

You only need to run `makecookie` once! After you run this, your cookie can be found in a file called `cookie.txt`.

In three of your four buffer attacks, your objective will be to make this cookie show up in places where it ordinarily would not. In two of those four attacks, you will accomplish this by supplying machine code instructions to the `buffer` program.

A problem with doing so is that Linux does not allow data on the program stack to be executed as machine instructions in an attempt to avoid such attacks, and Tom's servers just happen to use Linux. However, you tricked the programmers that wrote libraries security into moving the stack to a different, executable memory location. This means that (for this project) the instructions that you place on the stack can indeed be executed.

3 The Buffer Program

The `buffer` program reads a string from standard input. It does so with the function **`getbuf()`** defined below:

```
/* Buffer size for getbuf */
```

```
#define NORMAL_BUFFER_SIZE 32

int getbuf() {
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

The function **`Gets()`** is similar to the standard library function **`gets()`**—it reads a string from standard input (terminated by **`'\n'`** or **`end-of-file`**) and stores it (along with a null terminator) at the specified destination. In this code, you can see that the destination is an array `buf` having sufficient space for 32 characters.

`Gets()` (and **`gets()`**) grabs a string off the input stream and stores it into its destination address (in this case `buf`). However, **`Gets()`** has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

If the string typed by the user to **`getbuf()`** is no more than the size of the buffer, **`getbuf()`** will correctly return 1, as shown by the following execution example:

```
./buffer
Type string: I love CS 33.
Oops: getbuf returned 0x1
```

Typically an error occurs if a longer string is entered:

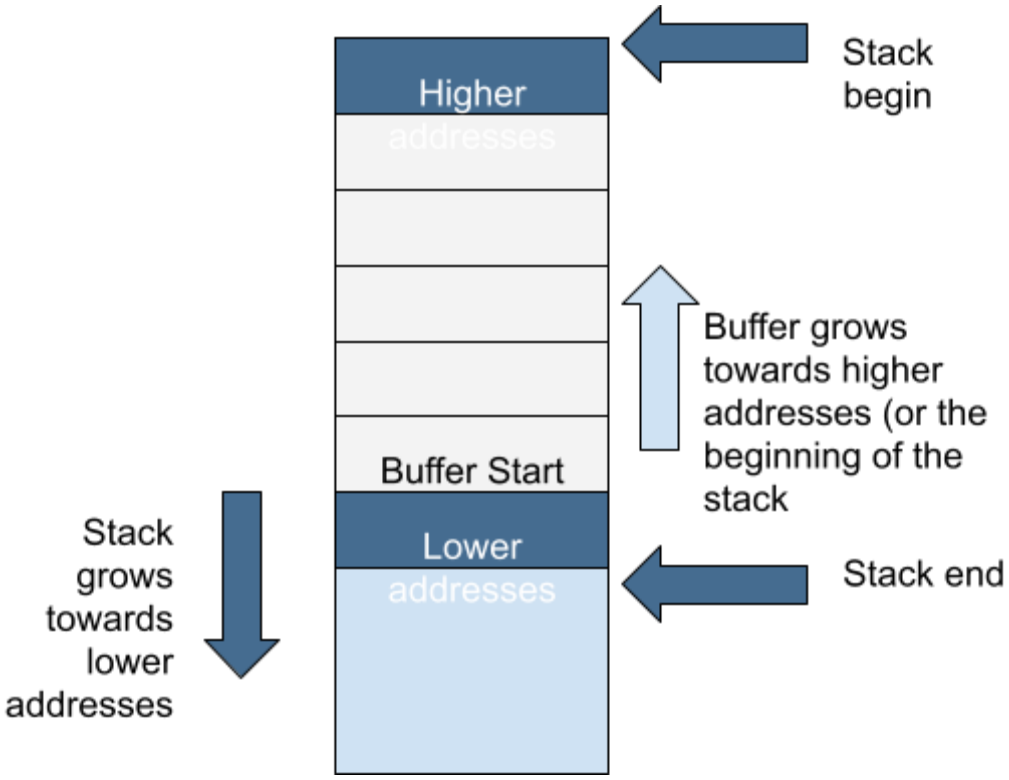
```
./buffer
Type string: It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!
```

As the error message indicates, overrunning the buffer typically causes the program state to be corrupted, leading to a memory access error. Your task is to be more clever with the strings you feed `buffer` so that it does more interesting things. These are called *exploit* strings.

`buffer` takes a few different command line arguments:

`-`**`h`**: Print list of possible command line arguments.
`-`**`n`**: Operate in “Nitro” mode, as is used in Level 4.

At this point, you should think about the x86-64 stack structure a bit and figure out what entries of the stack you will be targeting. You may also want to think about *exactly* why the last example created a segmentation fault, although this is less clear. Be aware that the buffer starts at the end of the stack (at lower memory address in the stack) and grows towards the beginning (towards higher addresses), while the stack itself grows from the beginning to the end.


Higher
addresses
Buffer Start
Lower
addresses
Stack
begin
Buffer grows
towards higher
addresses (or the
beginning of the
stack
Stack end
Stack
grows
towards
lower
addresses