1.

You are a modern day superhero, trying to hack into the supervillain's supercomputer. You have discovered that their supercomputer reads a string from standard input, using a function called "**Gets**" that is curiously identical to the one used in a class project from college, many years ago. The supercomputer uses **randomization**, and also marks the section of memory holding the stack as **non-executable**.

Thanks to the sacrifice of your trusty sidekicks, hotdog-man and one-punch-man, you managed to learn that the **buffer size of the "Gets" function is 32 bytes**. Furthermore, you learned the address and machine instructions of the following two functions:

```
0000000000401900 <boomBoomBOOM>:
  401900:   55                      push   %rbp
  401901:   48 89 e5                mov    %rsp,%rbp
  401904:   b8 48 89 c7 90          mov    $0x90c78948,%eax
  401909:   5d                      pop    %rbp
  40190a:   c3                      retq

000000000040190b <bangBangBANG>:
  40190b:   55                      push   %rbp
  40190c:   48 89 e5                mov    %rsp,%rbp
  40190f:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  401913:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  401917:   c7 00 58 90 90 c3       movl   $0xc3909058,(%rax)
  40191d:   90                      nop
  40191e:   5d                      pop    %rbp
  40191f:   c3                      retq
```

movq S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

In order to save your city, you need to call a function with the address **`0x400090`**, that takes the number "**12345**" as input. **What should your input string be**, in order to execute that function with the appropriate input?

2.

For one of your solutions in the attack lab, draw the state of the stack every time it changes. Draw an arrow for where %rsp points to. Also draw an arrow for where %rip points to.

Fun fact: Whatsapp was actually just hacked by a buffer overflow attack:
https://www.wired.com/story/whatsapp-hack-phone-call-voip-buffer-overflow/

3.

```
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

After compiling the program and running it, you get the output:

```
Hello, world.
Hello, world.
```

You run the program again and the output this time is:

```
Hello, wHello, woorld.
rld.
```

Explain this behavior.

(Source: http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall16/Lectures/openmp.html)

4.
Take a look at the following OpenMP usages.

a.
Is there a difference between the two following codes? We want func() to be called 10 times.

```
#pragma omp parallel num_threads(2)
{
      ...
      #pragma omp parallel for
      for (int i = 0; i < 10; i++)
      {
            func();
      }
}
Vs.
#pragma omp parallel num_threads(2)
{
      ...
      #pragma omp for
      for (int i = 0; i < 10; i++)
      {
            func();
      }
}
```

b.
What is the issue with the following code? What can we do instead?

```
#pragma omp parallel
{
      omp_set_num_threads(2);
      #pragma omp for
      for (int i = 0; i < 10; i++)
      {
            func();
      }
}
```

5.

Consider the following function. How might we optimize it using OpenMP?

```
void func3(double *arrayX, double *arrayY, double *weights,
           double *x_e, double *y_e, int n)
{
     double estimate_x=0.0;
     double estimate_y=0.0;
     int i;

     for(i = 0; i < n; i++){
           estimate_x += arrayX[i] * weights[i];
           estimate_y += arrayY[i] * weights[i];
     }

     *x_e = estimate_x;
     *y_e = estimate_y;

}
```

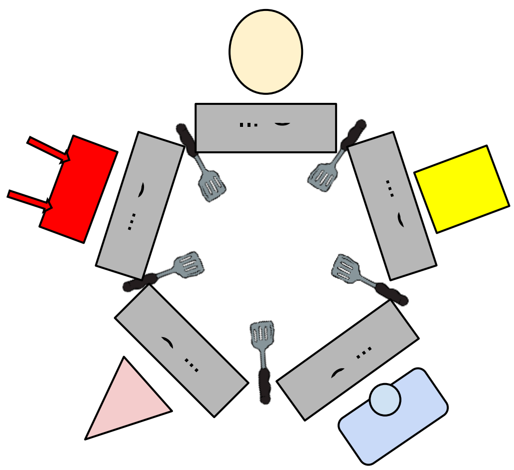
6. Extra.

a.
The four conditions under which deadlock occurs are:

1. Mutual Exclusion
2. Incremental (or partial) Allocation
3. No pre-emption
4. Circular Waiting

What do these conditions mean? In what ways (if at all) can these conditions be useful?

b.
Bored of blowing bubbles, Spongebob and 4 of his friends decide to make krabby patties instead. To make krabby patties, one needs 2 spatulas, both at the same time. However, they discover that they only have 5 spatulas total.



Each of Spongebob and his friends can only grab one spatula at a time, and can only grab spatulas to their left and right. All of them prefer to pick up the left spatula first, then the right. They refuse to forcefully take away spatulas from each other, lest they break their friendship, and will pick up a spatula only if it is not being held. Once they have even one spatula, they refuse to let go of it until they can make a krabby patty.

Is this situation considered a deadlock? Why or why not?
If so, how does it fit into the four conditions for deadlock? How can we resolve it?
If not, what about this situation helps Spongebob avoid deadlock?