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)

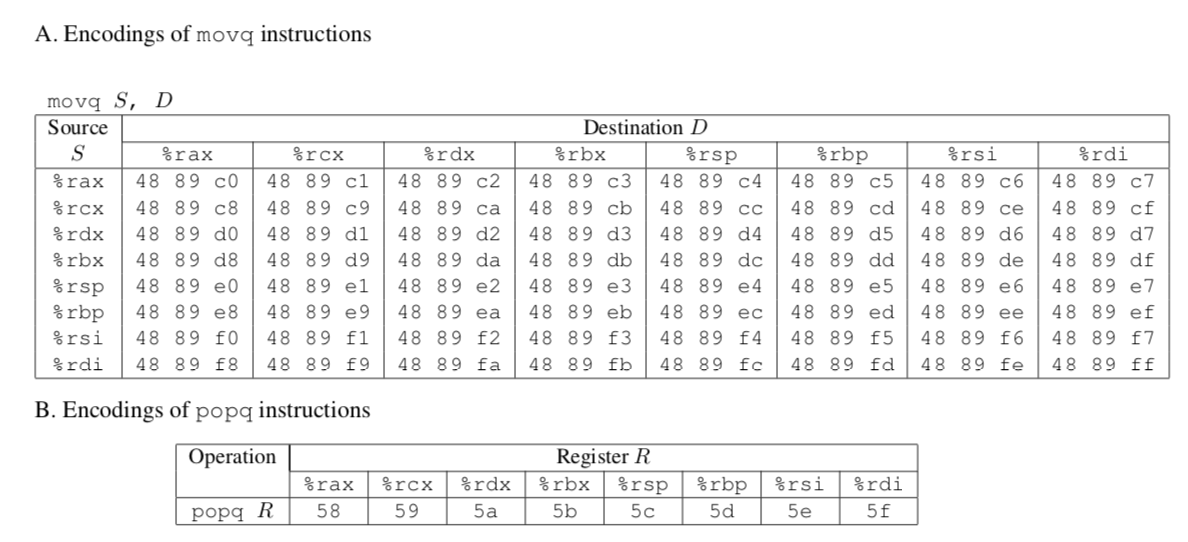
401913: 48 8b 45 f8 mov -0x8(%rbp),%rax

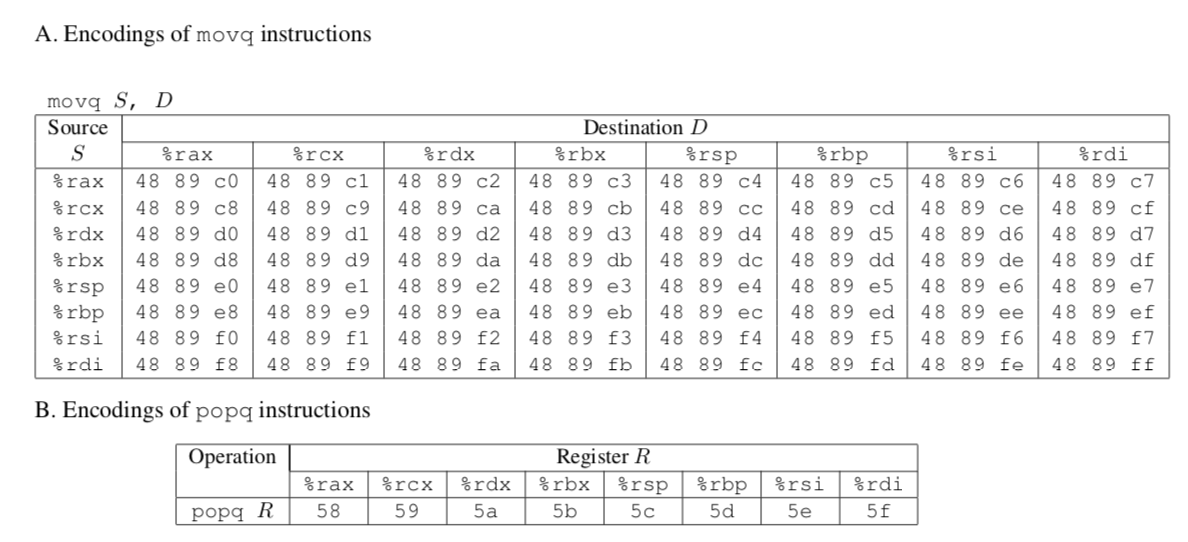
401917: c7 00 58 90 90 c3 movl $0xc3909058,(%rax)

40191d: 90 nop

40191e: 5d pop %rbp

40191f: c3 retq





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>)

The order in which threads run are non-deterministic. Thus, it cannot be assumed that the printf statements will be executed in sequential order – although this is what coincidentally happened in the first run. In the second run, both threads raced to gain access to the standard output shared resource, and this caused the “Hello, world.” Statements to be overlapped.

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();**

**}**

**}**

**The first version will create two threads initially, then upon reaching the #pragma omp parallel for, branch out so that each of the original two threads be split further to do this domain decomposition. Overall, there will be 20 loops, rather than 10. The second version is correct because the inner #pragma omp statement will not spawn a set of threads for each of the original two threads.**

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();

}

}

The number of threads cannot be changed within a parallel section. Move it out and join it with the #pragma omp parallel clause.

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;

**#pragma omp parallel for reduction(+:estimate\_x, estimate\_y)**

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?

Mutual exclusion means that two threads cannot both access a resource, such as a variable to modify. This can be useful when you want to ensure two threads don’t both modify a variable at the same time.

Incremental allocation is when a thread holds onto its own resource and “locks” it from use by other threads until it receives the next resource it requires. This is useful in allowing for concurrency, because the common resources between threads will be compartmentalized and one thread cannot control all resources and prevent other threads from using them.

No pre-emption means that no shared resources that are locked by one thread can be forcefully unlocked by another thread. This ensures that atomic operations are valid and not undermined.

Circular waiting may occur as a response to all of these conditions. A resource may require mutual exclusion, such as a current node pointer in a linked list. Incremental allocation would be implemented as a result, in order to only allow one thread to modify the current node pointer if another thread has been finished with it. Not allowing pre-emption will ensure that the node pointer cannot be forced out of the control of one thread and into another. Thus, if these conditions are implemented poorly, a shared resource can be held indefinitely by one thread, which is waiting for a different shared resource that is held indefinitely by another thread. Both threads are waiting for the same thing and will not give up their resource.

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?

Yes, this situation is considered a deadlock. Each of Spongebob and his friends can be considered a thread which has held on to its own shared resource, refusing to let go until being able to access a different shared resource. But that other shared resource is already locked, so nobody can do anything.

This can be resolved by not allowing each of Spongebob and his friends to grab a spatula at the same time. The spatulas can be better shared.