1. What is the value of y after both of the following operations?

x = x ^ (~y);

y = y ^ x;

~x

y = y ^ (x ^ (~y)) → (y ^ ~y) ^ x → 1s ^ x == ~x

After you plug in x, you can use the commutative and associative properties of XOR and do y^~y first which results in all 1s. x XORed with 1s flips its bits, thus ~x

Say x = 0111 and y is 1010

0111 ^ 0101 = 0010

1010^0010 = 1000 which is ~x

1. Given the following declarations, do the statements below always evaluate to true?

int x = foo();

int y = bar();

unsigned ux = cookie();

a.

x > ux ====> (~x+1) < 0

FALSE

Consider x = -1.

* The binary is all 1s, thus when you do ~(all 1s) it becomes all 0s.
  + Adding the 1 makes the value positive.

This is true for all negative x values since the sign bit will always be flipped to 0.

* So the ‘it follows’ is not true for all x > ux.

b.

ux - 2 >= -2 ====> ux <= 1

TRUE

If ux is 0

* it is comparing the unsigned values of -2 and -2, which are equal.

If ux is 1

* it is comparing the unsigned values of -1 and -2, which are Umax vs Umax -1.

2,3, etc

* aren’t true and ux can not be a negative value.

So, it follows that ux must be 0 or 1.

c.

(x^y)^x == (x+y)^((x+y)^y)

TRUE

Notice that both sides are of the form (A^y)^A

* For the left hand side, A = x
* For the right hand side, A = x+y

(A^y)^A is equivalent to y

* Thus, the equivalence simplifies to y == y
* Both sides of the equivalence are equal

d.

(x < 0) && (y < 0) == (x + y) < 0

FALSE

Say x == INTMin and y == INTMin.

* (x+y) would overflow.

1. char\*\* apple[5][9]; 360 bytes (8 \* 5 \* 9)

char\* banana[1][9]; 72 bytes (8 \* 1 \* 9)

char strawberry[4][2]; 8 bytes (1 \* 4 \* 2)

How many bytes of space would these declarations require?

1. Consider the following struct:

typedef struct {

char first;

int second;

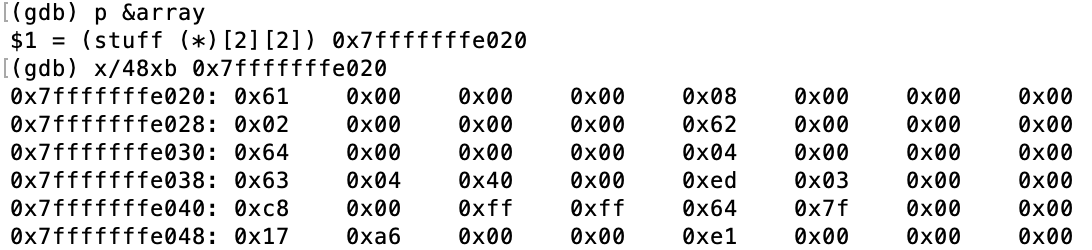
short third;

} stuff;

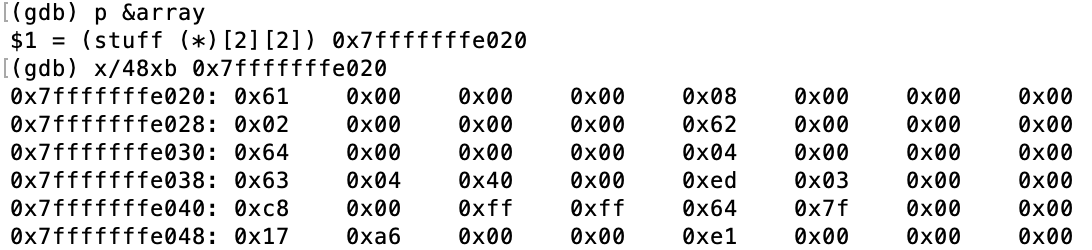
Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:

stuff array[2][2];

Using gdb we find the following information at a particular stage in the application:



And:



What is the value of

array[1][0].second

At this particular stage of the application?

i.e. what would be returned from the statement:

printf("%d\n", array[1][0].second);

**1005**

Because of alignment, each object of type “stuff” is 12 bytes.

Due to how arrays are stored in memory,

* The array is stored as:

array[0][0], array[0][1], array[1][0], array[1][1]

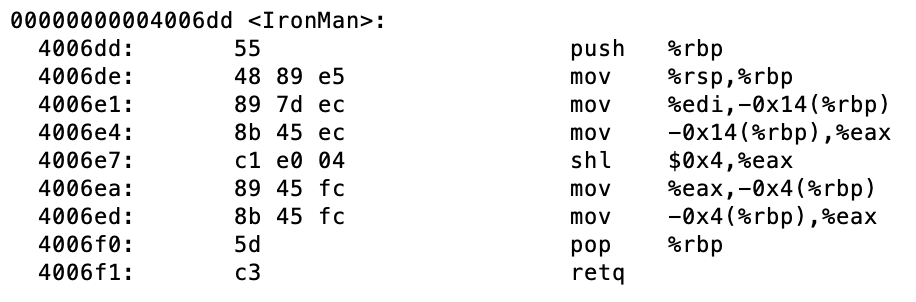
From the gdb output, we can tell that the array starts at 0x7fffffffe020

* array[1][0] is 0x7fffffffe038 to 0x7fffffffe043
  + Note: this is in hex, so 0x7fffffffe038 + 8 = 0x7fffffffe040

Second is an integer, and is the 5th to 8th byte of an object of type “stuff”

* These are bytes 0x7fffffffe03c to 0x7fffffffe03f
* They have the values 0xed, 0x03, 0x00, 0x00
* Since this system is little endian, the value is 0x000003ed
  + This is equivalent to 1005

1. The following is part of the result of the command ‘objdump -d’ on an executable



Say the declaration for the function IronMan was:

int IronMan(int scraps);

Given that the integer 23 was passed into the function, what is the return value?

368

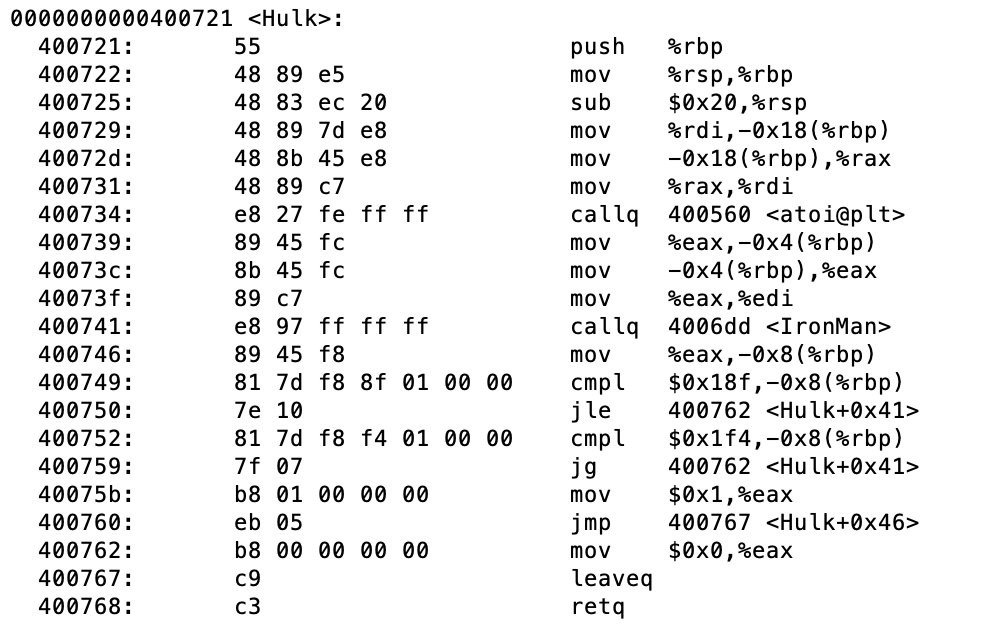
After instructions 0x4006e1 and 4006e4, the input (which was stored in %rdi) is now stored in %eax

Instructions 0x4006e7 then shifts %eax to the left by 4

* This is equivalent to multiply by 2^4, which is 16

23 \* 16 = 368

1. The following is a continuation from the previous problem:



Given that the function returns 1, what do we know about the value of %edi right before instruction 0x400741 is executed?

%edi is between 25 and 31

Since the function returns 1, we know that the jump instructions at 0x400750 and 0x400759 did not jump.

* From instructions 0x400749 and 0x400750
  + we know that we would have jumped if -0x8(%rbp) was less than or equal to 0x18f
  + Thus we know -0x8(%rbp) is greater than 0x18f, or 399
* From instructions 0x400752 and 0x400759
  + We know that we would have jumped if -0x8(%rbp) was greater than 0x1f4
  + Thus we know -0x8(%rbp) is less than or equal to 0x1f4, or 500
* Thus we know that -0x8(%rbp) is between 400 and 500, inclusive
  + Thus %eax is between 400 and 500, inclusive

From the previous question, we know that IronMan multiplies inputs by 16

* We also know that the function returns a value between 400 and 500 with input %rdi
* Reversing the function, we know the input must have been between 400/16 and 500/16

Thus we know that %rdi was between 25 and 31 right before the IronMan function call