Integer Overflow

Lab 2

Himanshu Sharma – 202117006

Task-1: With integers, you use in ordinary arithmetic, there are only a few pairs you can multiply together to get 10: 1 and 10, -1 and -10, 2 and 5, or -2 and -5. However, because int variables in C have a limited bit width, they behave somewhat differently. Explain how to find, and give, another pair of 32-bit ints which multiply together to get 10, even though they wouldn't as mathematical integers. There is a simple example you can write down (in hex) without any complicated calculation: it may be easiest to see if you think about the fact that multiplying by a power of two is equivalent to shifting left in binary.

Sol: In this task we have to multiply two numbers such that their product is 10. For the given question we have to use the property of integer overflow to get our result.

For all the question we will use following notation:

- **num1** = first number
- **num2** = second number
- **result** = required number
- **MOD = (2^32)** maximum 32-bit unsigned integer

We can derive a general equation:

(num1 * num2) % MOD = 10

Let's take: **num2 = 2** (easier to perform shift operation)

$$(num1 * 2)\ \%\ MOD = 10$$

$$num1 = \left(\frac{10}{2}\right)\%\ MOD$$

$$num1 = \left(\frac{MOD + 10}{2}\right)\ \%\ MOD$$

From the above equation, we can see that num1 can be obtained by dividing a number (MOD+10) by 2.

Let: K => MOD+10

K => (0x 01 00 00 00 00) + (0x 00 00 00 0A)

K => 0x 01 00 00 00 0A

Now dividing K by 2 (right shift 1 time), we get

Num1 => 0x 00 80 00 00 05

0000 0001	0000 0000	0000 0000	0000 0000	0000 1010	=>	0x 01 00 00 00 0A
	1000 0000	0000 0000	0000 0000	0000 0101	=>	0x 00 80 00 00 05

We can cross verify the solution:

- ⇨ Num1 * 2
- ⇨ (0x 00 80 00 00 05) * (0x 00 00 00 00 02)
- ⇨ (0x 01 00 00 00 0A) % MOD
- ⇨ (0x 00 00 00 00 0A)
- ⇨ (Hence verified)

Hence our result: num1 = (0x 80 00 00 05) and num2 = (0x 00 00 00 02)

Task-2: With normal integers there isn't any integer you can multiply by 13 to get 10. But again C ints are different. Explain how to find, and give, a 32-bit int which yields 10 when multiplied by 13. You can get the answer, just with a brute-force search through all 2^32 possibilities, but think of a cleverer approach.

Sol: Similar to the question 1 but with slightly different approach, we will take value which is greater than MOD and try to divide it with 13.

$$num1 = \left(\frac{MOD` + 10}{13}\right) \% MOD$$

Here we will take MOD+10 divide it by 13 and check if num1 satisfies the condition. If it satisfies that is answer and break the loop. However, if it didn't then, we will increase the value of 5th byte with value of 1 until we reach to the solution.

K^0 => 0x 01 00 00 00 0A

K^1 => 0x 02 00 00 00 0A

K^2 => 0x 03 00 00 00 0A

K^3 => 0x 04 00 00 00 0A

……

K^n => 0x FF 00 00 00 0A (and so on …)

As we know that if we multiply a number by 13, we cannot get 10 in normal conditions. Only if the product of number and 13 is large enough, it overflows to give result = 10.

So, by only increasing value of 5th byte, we greatly lower the number of iterations from 2^{32} to a significantly less value.

By following the above procedure, let's take

K^9 => 0x 09 00 00 00 0A

Divide K^9 by 13, we get

Num1 => K^9 / 13

Num1 => (0x 09 00 00 00 0A) / (0x 00 00 00 0D)

Num1 => 0x 00 B1 3B 13 B2

Checking if it satisfies given condition

- ⇨ Num1 * 13
- ⇨ (0x 00 B1 3B 13 B2) * (0x 00 00 00 00 0D)
- ⇨ (0x 09 00 00 00 0A) % MOD
- ⇨ (0x 00 00 00 00 0A) (10 in decimal)
- ⇨ (verified)

Hence our result: num1 = (0x B1 3B 13 B2)

Task-3: With integers, you use in math class, there are only a few pairs you can multiply together to get 18: 1 and 18, -1 and -18, 9 and 2, -9 and -2, 6 and 3, or -6 and -3. However, because int variables in C have a limited bit width, they behave somewhat differently. Explain how to find, and give, another pair of 32-bit ints which multiply together to get 18, even though they wouldn't as mathematical integers. There is a simple example you can write down (in hex) without any complicated calculation, if you use the fact that multiplying by a power of two is equivalent to shifting left in binary.

Sol: Similar to the first question, this question also asks for two different number such that their product is 18. We can use the integer overflow property to achieve this.

Here, $(num1 * num2) \% MOD = 18$

Let: $\boldsymbol{num2 = 2}$

$$num1 = \left(\frac{MOD + 18}{2}\right) \% MOD$$

For the purpose of solving the question we took a number greater than MOD and try to divide by 2 to check if it satisfy the condition

Let: K = MOD + 18

K = (0x 01 00 00 00 00) + (0x 00 00 00 12)

K = (0x 01 00 00 00 12)

If we divide this number by 2, we get

- ⇨ num1 = (K / 2)
- ⇨ num1 = (0x 01 00 00 00 12) / (0x 00 00 00 02)
- ⇨ **num1 = (0x 00 80 00 00 09)**

We can cross verify our result

- ⇨ (num1 * 2)
- ⇨ (0x 80 00 00 09) * (0x 00 00 00 02)
- ⇨ (0x 01 00 00 00 12) % MOD
- ⇨ (0x 00 00 00 00 12)
- ⇨ 18
- ⇨ (Hence verified)

Hence, num1 = (0x 80 00 00 09) and num2 = (0x 00 00 00 02)

Task-4: With normal integers there isn't any integer you can multiply by 7 to get 18. But again C ints are different. Explain how to find, and give, a 32-bit int which yields 18 when multiplied by 7. You can get the answer, just with a brute-force search through all 2^32 possibilities, but think of a cleverer approach.

Sol: Similar to the question 2, we will take value which is greater than MOD and try to divide it with 7.

$$(num1 * 7) \,\%\, MOD = 18$$

$$num1 = \left(\frac{MOD` + 18}{7}\right) \,\%\, MOD$$

Here we will take MOD+10 divide it by 7 and check if num1 satisfies the condition to give 18 as result. If it satisfies then that is answer and break the loop. However, if it did not then, we will increase the value of 5th byte with value of 1 until we reach to the solution.

K^0 => 0x 01 00 00 00 12

K^1 => 0x 02 00 00 00 12

K^2 => 0x 03 00 00 00 12

K^3 => 0x 04 00 00 00 12

……

K^n => 0x FF 00 00 00 12 (and so on …)

As we know that if we multiply a number (num1) by 7 we cannot get 18 in normal conditions. Only if the product of number (num1) and 7 is large enough, that will overflow result to 18.

So, by only increasing value of 5^{th} byte, we greatly lower the number of iterations from 2^{32} to a significantly less value.

By following the above procedure, let's take

K^6 => 0x 06 00 00 00 12

Divide K^6 by 7, we get

Num1 => K^6 / 7

Num1 => (0x 06 00 00 00 12) / (0x 00 00 00 07)

Num1 => 0x DB 6D B6 DE

Checking if it satisfies given condition (cross verify)

- ⇨ Num1 * 7
- ⇨ (0x 00 DB 6D B6 DE) * (0x 00 00 00 07)
- ⇨ (0x 06 00 00 00 12) % MOD
- ⇨ (0x 00 00 00 00 12) (10 in decimal)
- ⇨ (verified)

Hence our result: num1 = (0x DB 6D B6 DE)