Ignite CTF Write-Up

Russia - Painting

The image is a program of Piet. Piet is an esoteric programming language whose programs are bitmaps that look like abstract art. Google can help you find that the program actually prints Hello world!

India - MSRIT Campus

The image is directly taken from MSRIT's website and is visually identical to the source. There are many ways to solve this. But the first thing which you should try is the strings command:

```
 strings msrit.jpg
```

which print the strings of printable characters in a file. At the bottom of the output you would see flag.txt. Looks like flag.txt is inside the jpg. You can then try to open the image in a hex-editor (like Bless) and confirm it. You would also see "PK" near the flag.txt; most of the zip file signatures end with PK (0x4b50) after its inventor Phil Katz. By now it should be clear that there is a zip file inside the jpg.

```
mv msrit.jpg msrit.zip
unzip msrit.zip
```

Unzip ignores the jpg and extracts the zip content. Another way to solve it would be to use Foremost tool which recover files based on their headers, footers, and internal data structures. It would separate the jpg from the zip file.

China - Print Flag*

If you try to run the binary, it gives this:

```
The flag won't be printed so easily.
```

We have to disassemble it to see what is the binary internally doing. We can disassemble it using any tool/utility like IDA-Pro, objdump, gdb, Plasma etc. I used plasma and here is the output:

```
function main (.text) {
    0x4005b6: push rbp
    0x4005b7: rbp = rsp
    0x4005ba: rsp -= 32
    0x4005be: *(rbp - 20) = edi
    0x4005c1: *(rbp - 32) = rsi
    0x4005c5: *(rbp - 8) = '*'
    0x4005c9: *(rbp - 7) = '*'
    0x4005cd: *(rbp - 6) = '*'
    0x4005d1: *(rbp - 5) = '*'
    0x4005d5: *(rbp - 4) = '*'
    0x4005d9: *(rbp - 3) = '*'
    0x4005dd: *(rbp - 2) = '*'
    0x4005e1: *(rbp - 1) = '*'
    0x4005e5: r8d = *(rbp - 1)
    0x4005ea: edi = *(rbp - 2)
    0x4005ee: esi = *(rbp - 3)
    0x4005f2: r9d = *(rbp - 4)
    0x4005f7: r10d = *(rbp - 5)
    0x4005fc: ecx = *(rbp - 6)
    0x400600: edx = *(rbp - 7)
    0x400604: eax = *(rbp - 8)
    0x400608: rsp -= 8
    0x40060c: push r8
    0x40060e: push rdi
    0x40060f: push rsi
    0x400610: r8d = r10d
    0x400613: esi = eax
    0x400615: edi = 0x4006f8 "%c%c%c%c%c%c%c%c"
    0x40061a: eax = 0
    0x40061f: call printf
    0x400624: rsp += 32
    0x400628: rax = *(stdout_0)
    0x40062f: rdi = rax
    0x400632: call fflush
    0x400637: edi = 0x400709
    0x40063c: eax = 0
    0x400641: call printf
    0x400646: rax = *(stdout_0)
    0x40064d: rdi = rax
    0x400650: call fflush
    0x400655: edi = 0x400718 "The flag won't be printed so e..."
    0x40065a: eax = 0
    0x40065f: call printf
    0x400664: nop
    0x400665: leave
    0x400666: ret
}
```

I have replaced the flag with *. So, it is actually printing the flag! But why is it not shown in the output? Well, there are three printf calls and two fflush calls. Second printf is using the backspace character (\b) to delete the flag characters from std out.

An easy way to solve this problem would be to use ltrace which is a linux utility to display all the library calls made by the application.

```
__libc_start_main(0x4005b6, 1, 0x7fffc8a51ee8, 0x400670 <unfinished ...>
printf("%c%c%c%c%c%c%c%c", '*', '*', '*', '*', '*', '*', '*', '*')
= 8
fflush(0x7f9331048620********)
= 0
printf("\b\b\b\b\b\b\b\b")
= 8
fflush(0x7f93)
= 0
printf("The flag won't be printed so eas"...)                                   =
36
The flag won't be printed so easily.+++ exited (status 36) +++
```

Canada - Mystery Function

The Python script when executed gives OverflowError. Let's walkthrough the code to understand each component.

```python
def mystery_helper(a, b):
    if a < b:
        return a
    return mystery_helper(a - b, b)
```

Note: From now on I am going to be a bit mathematical rigorous as it might help you in solving even more complicated problems in an elegant way. And this is something which I found was lacking in the participants.

If we write the recurrence relation for the above function we get:

$$a_{n+1} = a_n - b_n$$
$$b_{n+1} = b_n$$

Note that b doesn't change so we can rewrite the above equations as:

$$b_{n+1} = b_n = b_{n-1} = b_{n-2} = \ldots = b_0 = b$$
$$a_{n+1} = a_n - b$$

By now it should be clear that we subtract b from a at each level of recursion until we get a value of a which is less than b.

$$a_{n+1} = a_{n-1} - b - b$$
$$a_{n+1} = a_{n-1} - 2b$$
$$a_{n+1} = a_{n-2} - b - 2b$$
$$a_{n+1} = a_{n-2} - 3b$$
$$\ldots$$
$$a_{n+1} = a_0 - (n+1)b$$

Let's assume that at k^{th} iteration a becomes less than b:

$$a_k = a_0 - kb$$
$$\implies a_0 = kb + a_k$$

This equation should look familiar to you; k is the quotient and a_k is the remainder. So this function just computes the remainder of a and b, wow!

Let's now look at mystery_function and replace mystery_helper with mod operator, just to make it easier to read.

```
def mystery_function(a, b):
    a = a % b
    for x in range(1, b):
        if (a * x) % b == 1:
            return x
```

What we are trying to find in this is x, such that:

$$ax \equiv 1(mod\ b)$$
$$or$$
$$ax = qb + 1$$

which is also known as modular multiplicative inverse. As it turns out, the above equation is specialised form of Bézout's identity:

$$a'x' + b'y' = gcd(a', b')$$

where

$$b' = -q$$
$$y' = b$$
$$gcd(a', b') = 1$$

Before we move forward and try to find the value of x, we should understand few properties of modular multiplicative inverse.

Lemma 1: There exist unique modular multiplicative inverse, $a^{-1} \in \mathbb{Z}_b = \{0, 1, 2, \ldots, b-1\}$, if $gcd(a, b) = 1$.

Proof: Let's assume there exist two solutions x_1 and x_2 such that $ax_1 \equiv 1(mod\ b)$ and $ax_2 \equiv 1(mod\ b)$.

By modulus subtraction property, we have $a(x_1 - x_2) \equiv 0(mod\ b)$. This implies $b \mid a(x_1 - x_2)$ (b divides $a(x_1 - x_2)$) . But since $gcd(a, b) = 1$, so $b \mid (x_1 - x_2)$. Also, x_1 and $x_2 \in \{0, 1, 2, \ldots, b-1\}$.

Clearly, $x_1 - x_2 < b$. But for $b \mid (x_1 - x_2)$ to be true the only possible way would be when $x_1 = x_2$ $\square$

Note: The reason behind why x should belong to $\mathbb{Z}_b$ lies in the definition of modular arithmetic. Consider for example, $3x \equiv 1(mod\ 11)$, $x = 4$ is the only solution which belongs to the set $\{0, 1, \ldots, 10\}$. But if we want to find all possible integer solutions, we just have to find the elements which are congruent to x modulo b, e.g. $15 \equiv 4(mod\ 11), -7 \equiv 4(mod\ 11)$ etc. So modular arithmetic limits the number of possible solutions but it can be easily extended to find infinetly many solutions.

Lemma 2: Modular multiplicative inverse exist iff $gcd(a, b) = 1$.

Proof: Hint - if $x = y + z$ and there exist d, such that $d \mid x$ and $d \mid y$ then $d \mid z$. $\square$

The two numbers in the Python script are indeed coprime ($gcd(a, b) = 1$), which can be verified by Wolframalpha.

```
print
mystery_function(903285192834718294712938047234989812342934812390481239481234908234897
23489712389471932852,90328519283471829471293804723498981234293481239048123948123490823
489723489712389471932847)
```

So now we know that the result is a unique number which lies in the range $[0, 90328519283471829471293804723498981234293481239048123948123490823489723489712389471932846]$. But how do we compute x ? Turns out there is a logarthimic time algorithm for computing modular multiplicative inverse called Extended Eucledian algorithm. The algorithm is described here and its iterative implementation is described here. The interested readers are encouraged to follow the pointers to understand the algorithm.

There is an easier way to get to the answer which was used by the participants during the contest. The relation between a and b :

$$\begin{gathered}
a = b + 5 \\
ax \equiv 1 (mod \;\; b) \\
\implies (b+5)x \equiv 1 (mod \;\; b) \\
\implies (bx + 5x) \equiv 1 (mod \;\; b) \\
\implies 5x \equiv 1 (mod \;\; b) \\
\implies 5x = nb + 1
\end{gathered}$$

So now we are left with the above equation and n can take any value as long as x is a valid modular multiplicative inverse. But here is an interesting observation: we want $nb + 1$ to be divisible by 5 and the last digit of b is 7 and if we multiply b by 2 and add 1, the last digit would become 5. This leads directly to the answer:

$$x = \frac{2b + 1}{5}$$

Australia - Poem*

This problem is based on poem code which is a cryptographic method and it was used in World War II. This blog explains the encryption technique in detail. The basic idea is as follows: Pick some words from the poem and use them as key and map each character of the words to a unique number. This unique number is used to decide the order of transposing the message. The first few characters in the cipher text are the meta data for finding which words were used as the key; each character represents the offset of the word, e.g. A for first, D for fourth etc. *Quick question: Is this a limitation of poem code that we can't use words whose offset is greater than 26? How do we fix it?*

In the problem, the meta data (first four characters) is twisted. Each character is one greater than the actual character, e.g. B for first, C for second, A for 26th. Another twist is that there are two cipher text concatenated and the first character in the meta data specifies the count of the number of messages.

A more systematic, algorithmic way to solve the problem, after you figure out that this isn't straight forward poem cipher, would be to use divide and conquer. Try all the permutations of fixed size cipher text (4, 8, 16), with or without meta data, to find dictionary words.

*** Unsolved during the contest**