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 prints the strings of printable characters in a file. At the bottom of the output you would see flag.txt. It looks like flag.txt is inside the jpg. You can then try to open the image in an 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), the initials of 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 the Foremost tool which recovers 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. The 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.

```
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 $x_1 = x_2$ □

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 $d \mid z$. □

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

```
print
mystery_function(903285192834718294712938047234989812342934812390481239481234908234897
2348971238947193285 2,903285192834718294712938047234989812342934812390481239481234908234
897234897123894719328 47)
```

So now we know that the result is a unique number which lies in the range $[0, 903285192834718294712938047234989812342934812390481239481234908234897234897123894719328 46]$. 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 :

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

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.

France - Lock Picker

The challenge here is to find the password to the zip file. A part of the question reads "However it's password protected. Crack the password to unzip and get your flag.", which actually has the password to the zip file in the question itself. The password being “unzip”. While that might not be directly obvious and easily overlooked, there are tools that can crack the password to zip files by trying a dictionary attack on it, like fcrackzip, etc. Once the zip file is extracted, you get a flag.txt file - which doesn’t seem to have any useful information other than random characters and the recognisable word “png” in it. If you run ‘file’ on the flag.txt file, you get an output like so:

```
file flag.txt
flag.txt: PNG image data, 330 x 330, 1-bit colormap, non-interlaced
```

So we can see that the file is a PNG image. Changing the extension from .txt to .png and opening the file should load it in your system’s default photo viewer and you’ll get a QR code, which on scanning with any QR code scanner leads to the flag.

Saudi Arabia - Yet another cipher!

This one is fairly obvious to those who are familiar with popular encryption techniques. The hash at the end has an ‘=‘ sign which is a give away most of the time that its base64 encoded. Decrypting the base64 encoded string online is as easy as googling for what base64 is. Once decrypted, you get a weird looking code that people familiar with morse code will recognise as morse code. If you decryt the morse code, you get the flag!

Madagascar - King Julian’s iPod

The file in the challenge is an mp3 file but if you try to play it, it yields pretty much nothing. If you open it in a text editor, you’ll see a whole lot of giberish, but you’ll also see a line like:

```
^@^@^@^@^@^@^@^@^@moc.putxen.www//:ptth6102MA
```

Which is actually 'AM2016http://www.nextup.com’ in reverse. This and the line in the question that reads 'King Julian being the backwards guy that he is’, should give you a clue that the file might be reversed. The hint for the question also tells you that king julian holds his iPod upside down, which is again a clue that the mp3 file might be reversed.

A simple few lines of python code can reverse the file for you like so:

```
f = open('mysong.mp3',"rb")
s = f.read()
f.close();
f = open('reversedmysong.mp3',"wb")
f.write(s[::-1])
f.close()
```

Playing reversedmysong.mp3, will not make any sense but it does start to sound like english. Playing it over and over might give you a hint that the audio is actually playing backwards, if the challenge description wasn't good enough in telling you that the song is reversed. So reversing the mp3 file either using an online tool, or something like audacity, will give you a proper mp3 file which says - "the flag is **********'.

United States - L0gin!

The challenge gives you an IP address and a port, and also says net cats are willing to join your team. The 'net cats' is a reference to netcat which is a unix tool to make TCP/UDP connections. You can connect to the remote server using netcat by running the following command:

```
nc <ip address> <port>
```

When you connect to the server and type anything the server responds with:

```
When in rome, be roman
wkh dqvzhu wr wklv fkdoohqjh lv pg5(vlpsoh)
```

The first line tells you that this is a 'ceasar' cipher. We can crack the cipher by guessing the key or using an online tool to guess and decrypt the key (google it).

The decrypted string tells you that the answer to the challenge is md5(simple). If you submit the md5 hash of 'simple' as the flag, it won't be accepted, however if you submit the same on the server by connecting to it as above, you will get no response - ie. the 'When in rome….' message doesn't appear anymore. This tells you that something is going on here. If you look at the hint to the question - it tells you that shells can be rewarding. Which basically means your reward is a shell. Which implies that when you submit the md5 hash of 'simple' on the server, you get a shell (basically allows you to enter commands like you

would at command prompt or the terminal).

Typing `ls` at the prompt will return something like:

```
flag.txt showme
```

Typing out `cat flag.txt` at the prompt will give you the flag.

Brazil - Hello Kitty!

The .chunk file is basically just to confuse those who might think its a well known extension. Changing this file to a .zip doesn't really work either. Opening the file in a hex editor will tell you that the file is basically a bunch of jpegs put together - because of the standard jpeg header bytes 'ffd8' being repeated multiple times. If you knew that, then you could just write a small script extract the files by reading from the jpeg begin byte (ffd8) to the ending byte and saving it to a separate file, and thus recovering all the images from the disk.chunk file. Another way to get all the jpegs is to use the foremost tool (http://foremost.sourceforge.net) which lets you extract files based on format signatures. The extracted files are basically pictures of cats, where one of the pictures has a bit.ly link to a youtube video. The youtube video is of the emperor who tells you the password.

If you open the disk.chunk file in a text editor like vi, you'll see the line:

```
The flag is md5 of the password from one of the cats!
```

So submitting the md5 hash of the password that the emperor reveals in the video will let you solve this challenge.

Greenland - Rouge Binary!*

This is definitely a difficult challenge for those who aren't familiar with disassembling binaries and assembly code, but its fairly reasonable in its difficulty. The challenge says that the server is running the same binary as the one provided. If we can dissassemble and figure out the password for the given binary, the same password will work on the server and give you the flag (connect to the server using netcat like in the 'L0gin!' challenge).

We can dissasemble the binary using plasma or any other disassembler. Plasma’s output is :

```
function main (.text) {
    0x400876: push rbp
    0x400877: rbp = rsp
    0x40087a: esi = str
    0x40087f: edi = _ZSt3cin
    0x400884: call _ZStrsIcSt11char_traitsIcEERSt13basic_istreamIT_T0_ES6_PS3_
    0x400889: edi = str
    0x40088e: call strlen
    # 0x400893: cmp rax, 0xa
    # 0x400897: jne 0x400981
    if (rax == '\n') {
        0x40089d: edx = (zero ext) *(str)
        0x4008a4: eax = (zero ext) *(0x6012a9)
        # 0x4008ab: cmp dl, al
        # 0x4008ad: jne 0x400972
        if (dl == al) {
            0x4008b3: edx = (zero ext) *(0x6012a7)
            0x4008ba: eax = (zero ext) *(0x6012a8)
            # 0x4008c1: cmp dl, al
            # 0x4008c3: jne 0x400972
            and if (dl == al)
            0x4008c9: edx = (zero ext) *(0x6012a9)
            0x4008d0: eax = (zero ext) *(0x6012a8)
            # 0x4008d7: cmp dl, al
            # 0x4008d9: je 0x400972
            and if (dl != al)
            0x4008df: eax = (zero ext) *(0x6012a1)
            0x4008e6: edx = al
            0x4008e9: eax = (zero ext) *(0x6012a2)
            0x4008f0: eax = al
            0x4008f3: ecx = rdx + rax
            0x4008f6: eax = (zero ext) *(0x6012a6)
            0x4008fd: edx = al
            0x400900: eax = (zero ext) *(0x6012a7)
            0x400907: eax = al
            0x40090a: eax += edx
            # 0x40090c: cmp ecx, eax
            # 0x40090e: jne 0x400961
            if (ecx == eax) {
                0x400910: eax = (zero ext) *(0x6012a3)
                0x400917: edx = al
                0x40091a: eax = (zero ext) *(0x6012a4)
                0x400921: al cmp '\x01'
                0x400923: sete al
                0x400926: eax = (zero ext) al
                # 0x400929: cmp edx, eax
                # 0x40092b: je 0x400961
                and if (edx != eax)
                0x40092d: eax = (zero ext) *(0x6012a4)
                0x400934: edx = al
                0x400937: eax = (zero ext) *(0x6012a5)
                0x40093e: eax = al
                0x400941: eax *= edx
```

```
                # 0x400944: cmp eax, 0x11c6
                # 0x400949: jne 0x400961
                and if (eax == 4550)
                0x40094b: esi = 0x400a74 "You Got it!"
                0x400950: edi = _ZSt4cout_0
                0x400955: call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
                0x40095a: eax = 0
                0x40095f: jmp ret_0x400995
            } else {
                0x400961: esi = 0x400a80 "Good Try\n"
                0x400966: edi = _ZSt4cout_0
                0x40096b: call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
                0x400970: jmp 0x400981
            }
        } else {
            0x400972: esi = 0x400a8a "Keep trying\n"
            0x400977: edi = _ZSt4cout_0
            0x40097c: call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
            goto 0x400981
        }
    } else {
        0x400981: esi = 0x400a97 "FAIL! :P\n"
        0x400986: edi = _ZSt4cout_0
        0x40098b: call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
        0x400990: eax = 0
    }
    ret_0x400995:
    0x400995: pop rbp
    0x400996: ret
}
```

We can see that there are a bunch of conditions and finally one of them leads to printing of the message ‘You’ve got it!’. If we can craft an input that satisfies all the conditions to get the binary to print out you’ve got it, we can use the same input on the server to get the flag.

Lets take this one condition at a time:

The first condition is here:

0x40088e: call strlen

0x400893: cmp rax, 0xa

0x400897: jne 0x400981

This is basically calling strlen on the input - and comparing the result of strlen with 0xa, which basically translates in english to check if the string entered is 10 characters in length. If the string isn't 10 characters in length, it jumps to 0x400981 -which is the starting address of the code that prints out the message: 'FAIL'

The second condition and the associated explanations are:

0x40089d: edx = (zero ext) *(str) ——————> load the first character of the string in dx register

0x4008a4: eax = (zero ext) *(0x6012a9) —> load the character at index 9 of str -ie last character to ax register

0x4008ab: cmp dl, al ———> compare the characters (al and dl as each character is only 8 bytes)

0x4008ad: jne 0x400972 —— > if not equal jump to the 'Keep Trying!' message

It basically checks if str[0] == str[9]

The next condition is similar to the one above, but checks for str[7] == str[8]

0x4008b3: edx = (zero ext) *(0x6012a7)

0x4008ba: eax = (zero ext) *(0x6012a8)

0x4008c1: cmp dl, al

0x4008c3: jne 0x400972

The next condition is also similar to the one above, but checks for str[8]!=str[9]

0x4008c9: edx = (zero ext) *(0x6012a9)

0x4008d0: eax = (zero ext) *(0x6012a8)

0x4008d7: cmp dl, al

0x4008d9: je 0x400972

The next condition, which is a little more involved is as follows:

0x4008df: eax = (zero ext) *(0x6012a1) ——> Load str[1]

0x4008e6: edx = al —> copy str[1] to dx register

0x4008e9: eax = (zero ext) *(0x6012a2) —> Load str[2]

0x4008f0: eax = al —> set ax register to str[2]

0x4008f3: ecx = rdx + rax —> add str[1] and str[2] and store in ecx

0x4008f6: eax = (zero ext) *(0x6012a6) —> load str[6]

0x4008fd: edx = al —> move str[6] to dx register

0x400900: eax = (zero ext) *(0x6012a7) —> load str[7]

0x400907: eax = al —> set ax register to str[7]

0x40090a: eax += edx —> Add str[7] and str[6] and store in eax

0x40090c: cmp ecx, eax —> basically compare str[1]+str[2] and str[6]+str[7]

0x40090e: jne 0x400961 —> If they are not equal jump to ‘Good try’ Message

The next condition is as follows:

0x400910: eax = (zero ext) *(0x6012a3) —> Load str[3]

0x400917: edx = al —> copy str[3] to dx register

0x40091a: eax = (zero ext) *(0x6012a4) —> load str[4]

0x400921: al cmp '\x01’ —> compare al register and 0x1 ie check if al register is all 1s

0x400923: sete al —> if previous compare was equal, set al to 1, otherwise set al to 0.

0x400926: eax = (zero ext) al -> copy al to ax register

0x400929: cmp edx, eax —> compare str[3] & eax register

0x40092b: je 0x400961 —> if equal jump to ‘Good try’ message, otherwise continue execution.

What the above lines of code did, is basically a bit wise XOR between str[3] & str[4] and compared the result with 0x1 ie all 1s => implies that no two corresponding bits between str[3] and str[4] can be equal.

For example if str[3] = ‘A’ and str[4] = ‘B’, then the result of str[3]\^str[4] == 1, so the condition is satisfied, ie. the code doesn’t jump to good try but continues.

The last and final condition is:

0x40092d: eax = (zero ext) *(0x6012a4) —> load str[4] to eax

0x400934: edx = al —> copy str[4] to dx register

0x400937: eax = (zero ext) *(0x6012a5) —> load str[5] to eax

0x40093e: eax = al —> copy al register to eax

0x400941: eax *= edx —> multiply str[4] and str[5] basically str[4]*str[5] and store the product in eax

0x400944: cmp eax, 0x11c6 —> check if the result is 0x11c6 or 4550

0x400949: jne 0x400961 —> if the product isn't 4550 jump to good try

if the condition is satisfied ie. the product is indeed 4550, the code enters the block that prints 'You've got it', so if we can craft an input that meets all these conditions, then the code will print 'you've got it' and if you give the same input on the server, it prints out the flag.

*** Unsolved during the contest**