# Reversing Irreversible Image Transformations

CTF Write-up

TTM4536
Ethical Hacking

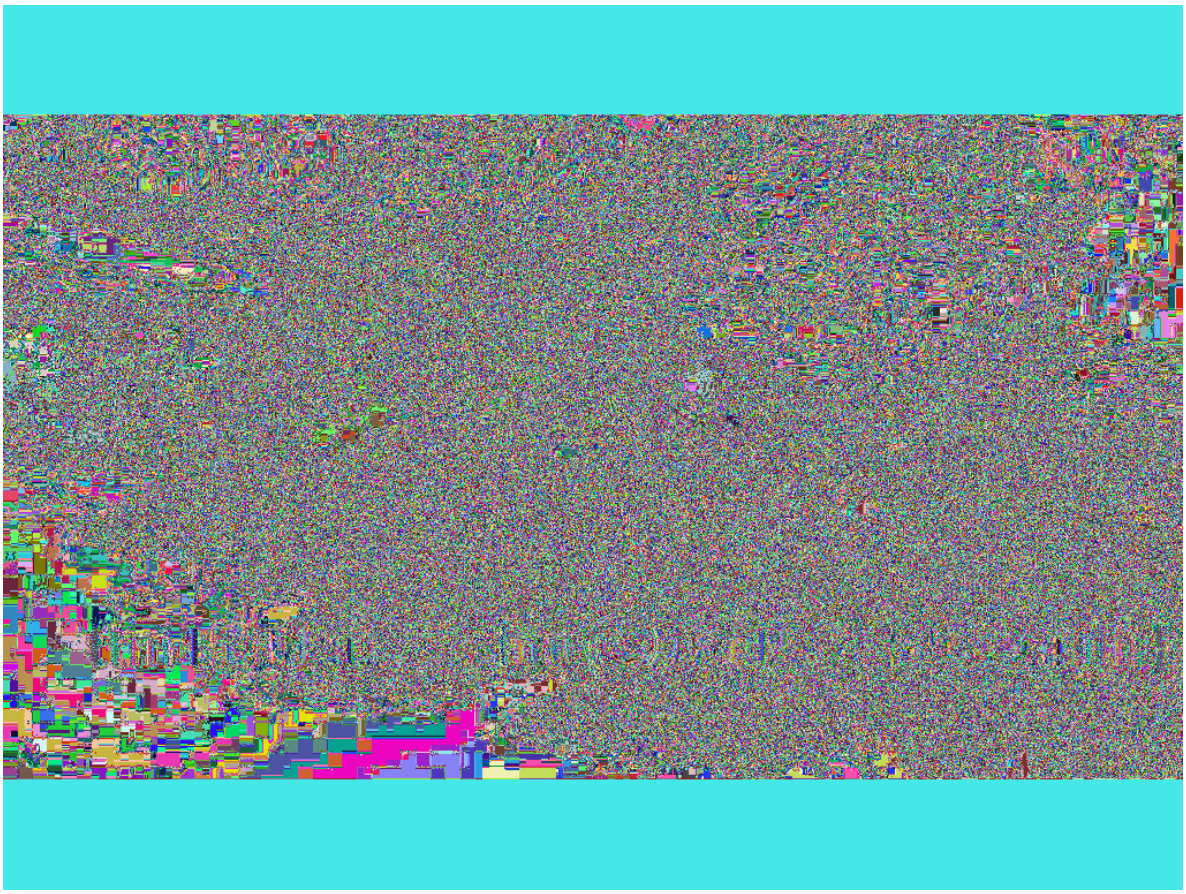
Halvor Bakken Smedås

November 2019

This write-up is part of the TTM4526 - Ethical Hacking held at NTNU, 2019
It's the sixth CTF locally held on the facebook CTF platform — Country: Malaysia
All code included is written in Python 3.6

## Contents

# 1 The challenge



This is what I was met with when I first started the challenge; a bitmap picture file, and a text-file with the `.py` extension

```
out1_e2ccdbfd607c147695bf5d733c5837e7.bmp

transform_8c0f9f59eb5dbf67e03748af36d55258.py
```

Now, normally when faced with an image looking like this, I'd immediately think *"well this is obviously corrupt and broken beyond repair"*... Naturally, considering this was a CTF-challenge, the next thought was *"Okay, so I'll need to repair it!"*. This thought only grew stronger after having looked at it for a little while; either my eyes were deceiving me, or I was seeing something resembling text in the nether region of the image.

At that point, I picked up my laptop and tried turning the screen into all thinkable angles, trying to decipher the "text" right before my eyes. It's got to be said: I opened the image in a couple of image processing applications as well, trying to see if just adjusting the pixels by some filters would grant me the solution I was so eagerly hoping to find (you never know, a CTF-challenge solved quick and dirty and may yet grant you more points than a nice and clean solution...), but again with little to no luck.

It was infuriating! *"The solution is right there, yet I cannot see what it is!"* — At least it was my assumption that what I was looking at was the solution. This later on turned out to be true; the solution is right there.

But let's not get ahead of ourselves! There was still a file I hadn't looked at — the text-file.

The text-file, as its file-extension would have it, was a python-program. Not only that, but it was a program that shed some light on what the image I had been looking at from every angle was.

```
1  from PIL import Image
2
3  import random
4  import sys
5
6  def qq(x, y):
7      return (2 * x + 3 * y + 29) % 256
8
9  def transform(pixelinfo):
10     pixelreverse = [pixelinfo[len(pixelinfo)-1-i] for i in range(len(pixelinfo))]
11     out = [pixelinfo[i] for i in range(len(pixelinfo))]
12     for i in range(len(pixelinfo)):
13         out[0] = qq(pixelreverse[i], out[0])
14         for j in range(1,len(pixelinfo)):
15             out[j] = qq(out[j-1], out[j])
16     return out
17
18
19 image = Image.open(sys.argv[1])
20 outfile1 = Image.new(image.mode, image.size)
21
22 for x in range(0, image.size[0]):
23     for y in range(0, image.size[1]):
24         sourcepixel = list(image.getpixel((x, y)))
25         tran = transform(sourcepixel)
26         outfile1.putpixel((x, y), tuple(tran))
27
28 outfile1.save('out1.bmp')
```

The first hint was in the last line of the program — it would make a new file called `out1.bmp`... Remember what the image file was called? — `out1_e2ccdbfd607c147695bf5d733c5837e7.bmp` Maybe, just maybe, this file was a result of this program?

Looking further into the code, this seemed more and more plausible. It reads an image-file (line 19), then iteratively goes through each and every pixel of said image (22-23), which it then transforms by the `transform`-function (9-16) which is modifying each color of the RGB-pixel through the `qq`-function.

From there on, my working hypothesis was that the flag was hidden as part of the image, and that the image was a result of the python program, making my task pretty clear: Reverse the transformation and apply it to the image to get the original image out.

---

Sounds simple enough, right? — Wrong! There is one huge problem — We're working in a discrete space; with pixels of $3 \times 8$-bit precision (RGB), meaning there are finite values a pixel may have. The transformation has taken this into account, and so it's defined to wrap over-/underflowing pixel values back around using the modulus operator.

Let's take a closer look at the piece of code we need to reverse:

```
6  def qq(x, y):
7      return (2 * x + 3 * y + 29) % 256
8
9  def transform(pixelinfo):
10     pixelreverse = [pixelinfo[len(pixelinfo)-1-i] for i in range(len(pixelinfo))]
11     out = [pixelinfo[i] for i in range(len(pixelinfo))]
12     for i in range(len(pixelinfo)):
13         out[0] = qq(pixelreverse[i], out[0])
14         for j in range(1,len(pixelinfo)):
15             out[j] = qq(out[j-1], out[j])
16     return out
```

Listing 1: The transformation applied by the python program

| | |
|---|---|
| `transform` takes one pixel and transforms it in 3 passes, iteratively by the original channel-values $(r, g, b)$ in the reverse order. Where the result of the transformation ultimately is the pixel $(r_3, g_3, b_3)$ | $r_1 = \texttt{qq}(b,\ r\ )$<br>$g_1 = \texttt{qq}(r_1, g\ )$<br>$b_1 = \texttt{qq}(g_1, b\ )$<br>$r_2 = \texttt{qq}(g,\ r_1\ )$<br>$g_2 = \texttt{qq}(r_2, g_1\ )$<br>$b_2 = \texttt{qq}(g_2, b_1\ )$<br>$r_3 = \texttt{qq}(r,\ r_2\ )$<br>$g_3 = \texttt{qq}(r_3, g_2\ )$<br>$b_3 = \texttt{qq}(g_3, b_2\ )$ |

## 2 The modulus operator

While other arithmetic operators to my knowledge generally has nice inversions (the whole field of algebra is based around this), the modulus operator does not. Intuitively that makes sense right? — If I were to say $15\%10 = 5$, then swap out the 15 for an X ($X\%10 = 5$), there is no way of knowing that X is supposed to be 15, it could just as well be 25 or 35, or literally any other number in base 10 that ends in a 5. The only thing we know is that the X will have a value of has a remainder of 5 when divided by 10, but there are infinitely many solutions to that equation, so how can we sensibly inversly solve it for an X that makes sense in the transformation function `qq`?

## 3 Reversing the irreversible — Approaches

*"So how then, do I go about solving this task, when the function is irreversible?"* — That's what I thought at this point. It turns out that there are a couple of ways to approach this problem:

### 3.1 The mathematical approach

Scratching my head looking for ways to surpass this problem of irreversibility, I discovered that there is a concept relating to the modulus operator called *equivalence classes* and *residue systems* both of which describe the properties how modulus equations have several equally valid solutions. Reading up on these for a while I found that there is something called *least positive residue*, which essentially is the set of numbers $n \in \{0, 1, 2, ..., m-1\}$ of $n \bmod m = n$ — i.e. all positive numbers below and up to the divisor number of the modulo operation, which might yet give me the results I want.

At this point the math really got out of hand for me. The expression the transformation (see listing 1) was constructing was very long — I needed them in their full form to be able reverse them, so I got wrote a script to generate them for me:

```
1  def str_qq(x, y):
2      return "((2 * {} + 3 * {} + 29) mod 256)".format(x, y)
3
4  def print_transformation():
5      pixelinfo = ["x", "y", "z"]
6      pixelreverse = [pixelinfo[len(pixelinfo)-1-i] for i in range(len(pixelinfo))]
7      out = [pixelinfo[i] for i in range(len(pixelinfo))]
8      for i in range(len(pixelinfo)):
9          out[0] = str_qq(pixelreverse[i], out[0])
10         for j in range(1, len(pixelinfo)):
11             out[j] = str_qq(out[j - 1], out[j])
12     for l in out:
13         print(l) # Print the final transformation applied to each channel as an expression
14
15 if __name__ == '__main__':
16     print_transformation()
```

The expressions resulting from this were super long, so I consulted a trusty friend: **WolframAlpha**

This helped me compress the expressions quite a bit, a pixel $p$ transformed into $p'$ by these changes in colors:

$$r_{p'} = (29r_p + 6g_p + 18b_p + 121) \bmod 256$$
$$g_{p'} = (166 * r_p + 51 * g_p + 108 * b_p + 45) \bmod 256$$
$$b_{p'} = (656r_p + 234g_p + 459b_p + 45) \bmod 256$$

Getting to this point didn't help me much, but I've heard from others that at this point you would need to use the equations as a matrix, and take its inverse to arrive at an actual inversion technique.

...

Instead, I went on to explore other solutions where I might have a better understanding of what I was doing.

## 3.2 Lookup-table

I've mentioned before that this problem is a discrete-space problem. Why not utilize this fact? It might be feasible to generalize the problem of transforming one pixel to another to apply across the entire discrete color space, and by doing so find which pixel values relate to which transformed pixel values.

A lookup-table of sorts, mapping all transformed $p'$ to their respective original $p$

That way we can lookup each pixel of the transformed image in order to go back to the original.

## 3.3 Hold on a minute... What about brute-force?

Before we get too far into this, there's one potential solution that is not yet discussed, and before starting to implement anything else at all it it should be considered!

To some degree, the image could be compared to a combination lock with three wheels (RGB), where each wheel goes from $0 \rightarrow 255$. In that sense, we could iterate through possible solutions by turning these wheels around, but how do we do that? — Well, we could construct a new transformation function that churns each pixel channel one by one, like a typical combination-lock brute-force "attack".

Instead, I want to argue that maybe, just maybe, the transformation-function we already have might do the trick. Considering that the transformation linearly scales from the image's pixel values, it should, when wrapped around (by the modulo operator) a couple of times land us somewhere close to the original.

This is where the analogy of the image being a combination lock ends, because we only need the image to resemble the original, as long as its somewhere close to it, the text should be decipherable.

After all, why do the work of implementing some complex inverse transformation, when the transformation you already have might be enough?

# 4 Solution

My ultimate solution, interestingly was a brute-force one. After having tried to come up with a solution to inverting the results of `transform` and `qq`. I found that there were two other approaches I could try instead, both essentially ignoring the mathematical aspects of doing a reversal:

- One approach that really seemed promising was simply to generate all possible pixels in color space (i.e. all possible combinations of red, green and blue), and transform them by the `transform`-function. That way, if I coupled the original value, and the transformed value, I could do a lookup over each pixel in the image. Looking at the non-transformed pixel of the pair found in such a lookup, I would get the original pixel out. At least this was my hypothesis.
- Another approach I thought of was that the transformation may, upon iteration over time, lead back to the original image, or at least an image resembling the original close enough that the flag-text might be visible.

*"I might as well keep a brute-force calculation run while I try to figure out how to do it in any of the other ways"* I thought to myself. It didn't take more than a couple of minutes before I had the code stitched together, and let it churn away.

```
if __name__ == '__main__':
    image = Image.open('out.bmp')
    outfile = Image.new(image.mode, image.size)
    iteration = 1
    while True:  # Just keep going forever, I'll cancel it if I see any results
        for p in [(x, y) for x in range(0, image.size[0]) for y in range(0, image.size[1])]:
            sourcepixel = list(image.getpixel(p))
            tran = transform(sourcepixel)
            outfile.putpixel(p, tuple(tran))
        outfile.save("out/brute_force/brute_force{}.bmp".format(iteration))
        image = outfile  # Start next iteration from last output
        iteration += 1
```

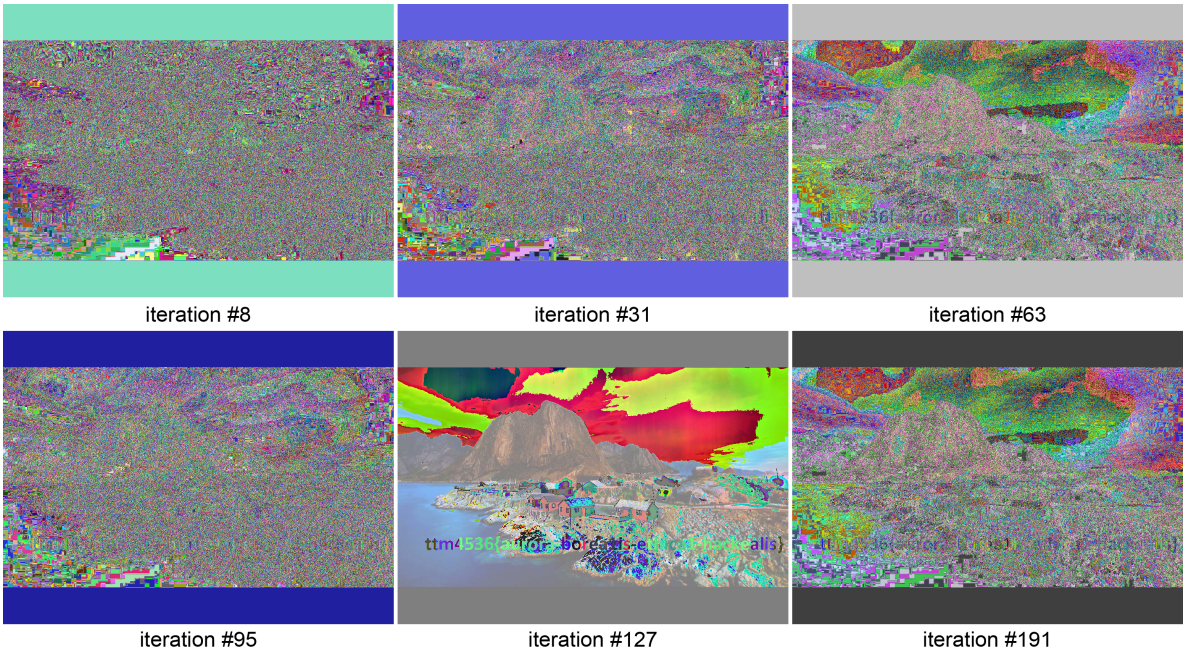
I'll come back to the results of this in a bit...

After starting my brute-force loop, I started looking at how I might implement a reversing transformation using a lookup-table. This too turned out to be quite a simple task.

```
transform_table = {} # Lookup-dictionary

def fill_table():  # Fill a lookup-table/mapping from transformed values to original values
    for b in range(0, 255):
        for g in range(0, 255):
            for r in range(0, 255):
                # Store an entry with the transformed pixel as key, and the original as value
                transform_table[tuple(transform([r, g, b]))] = (r, g, b)

def reverse(pixel):
    return transform_table[pixel]

if __name__ == '__main__':
    image = Image.open("out.bmp")
    fill_table()
    transformed = Image.new(image.mode, image.size)

    for p in [(x, y) for x in range(0, image.size[0]) for y in range(0, image.size[1])]:
        pixel = image.getpixel(p)
        transformed.putpixel(p, reverse(pixel))

    transformed.save('reversed.bmp')
```

Listing 2: Lookup-table implementation

By the time I had completed writing this code, the brute-force loop had made 211 iterations each resulting in new images, some of which really stuck out (but don't worry, there are some more on this approach in Section 5):

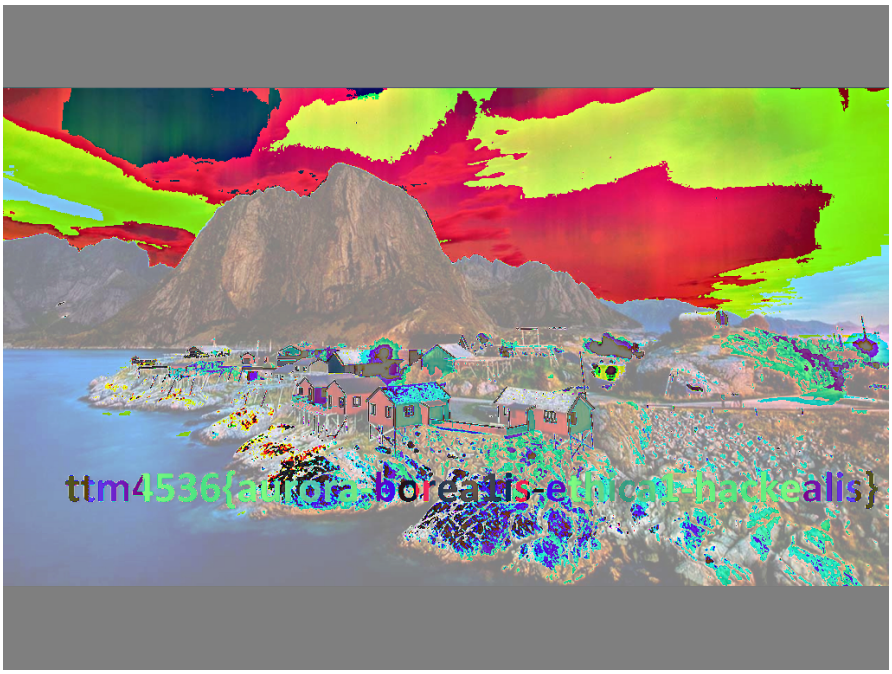



A selection of interesting outcomes from brute-force looping

Lo and behold — The flag is revealed!

While it still is pretty difficult to see, after 127 iterations of transformations, the flag is clear enough to be read. Looking at the others, the flag is actually almost readable in the 63rd iteration too, at least some of the letters, so at that point a solution could've been to just try each and every character from the ascii-set for the remaining unclear characters (the only problem being that it's difficult to see how many missing characters there were).

Let's have a closer look at iteration 127:



Now, it's not easy, but you can distinguish the flag from the rest of the image, and find that is says: `ttm4536{aurora-borea1is-ethica1-hackealis}` (note the 1's, I didn't at first...)

# 5 Post mortem additions...

## 5.1 Lookup-table

A major problem with a lookup-solution is the memory footprint of a lookup table. Just think about it — We're essentially creating $256^3$ pixels, which each probably will consist of three integers, which on typical systems are 32 bits, or 4 bytes, making one pixel 12 bytes, resulting in a lookup-table of size $12_{\mathrm{B}} \times 256^3 = 201\ 326\ 592_{\mathrm{B}} = 192_{\mathrm{MB}}$, wich on todays RAM chips, aren't that big of a deal. But it begs the question of what happens when this is applied in a scaled problem.

The answer is that it doesn't really — depending on which type of scaling is done. What I mean is, it works just as fine when applying this solution to an image of twice, thrice or even 100 times the size of this image. As the image size doesn't really affect the lookup-table's size at all, only it's color depth does...

If we were operating with an image with 16 bits or even 32 bits precision for each of the color channels, the sizes might would be overwhelming. Most images only operate with 8-bit precision however, so I won't go into details on 16- and 32bit color precision-images now. I'll rather talk about the more common case: *RGBA-images* — images with an additional channel (typically used for transparency). With the addition of another 8-bit channel, we'll already start seing major problems in terms of memory footprint, indicating that this is not a scalable solution.

with 4 channels there are a total of $256^4$ variations of pixels. Each (without any pixel packing) will be of $16_{\mathrm{B}}$ size (with packing however one could fit an RGBA-pixel into one 32bit integer, as each color technically only need 8 bits...so there is optimization potential there too).

$16_{\mathrm{B}} \times 256^4 = 68\ 719\ 476\ 736_{\mathrm{B}} = 64_{\mathrm{GB}}$ (with pixel packing it would roughly be a 16th of that, i.e. $4_{\mathrm{GB}}$, which is not too bad)

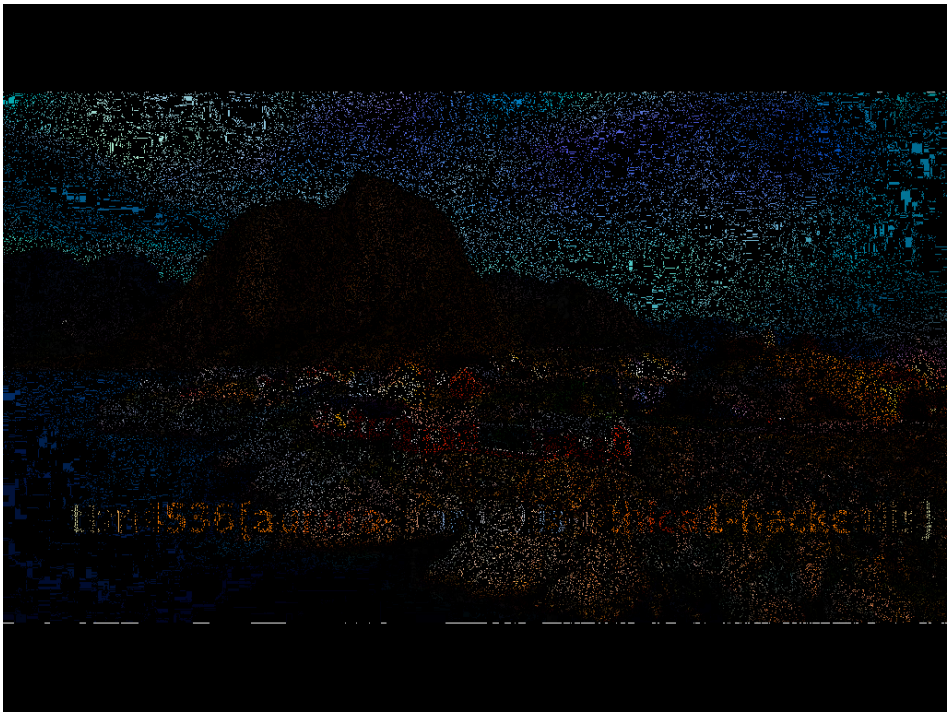
---

We don't need to keep all of the pixels in the lookup-table though — There is only a certain selection of the possible pixel variants the transformed image is using, so why keep the rest of them?

This simple idea allows us to optimize the previously defined lookup-table implementation: (see Listing 2)

```
1  transform_table = {} # Lookup-dictionary
2
3  def fill_table():
4      for b in range(0, 255, 2):
5          for g in range(0, 255, 2):
6              for r in range(0, 255, 2):
7                  # Store an entry with the transformed pixel as key, and the original as value
8                  transform_table[tuple(transform ([r, g, b]))] = (r, g, b)
9
10 def reverse(pixel):
11     if pixel in transform_table: # we're no longer sure each lookup grants a result
12         return transform_table[pixel]
13     else:
14         return 0, 0, 0  # return black if there is no lookup-result
```

Listing 3: Naïve optimization of the lookup-reversal; only sample a subset of all pixels and hope it grants you enough of the original pixels



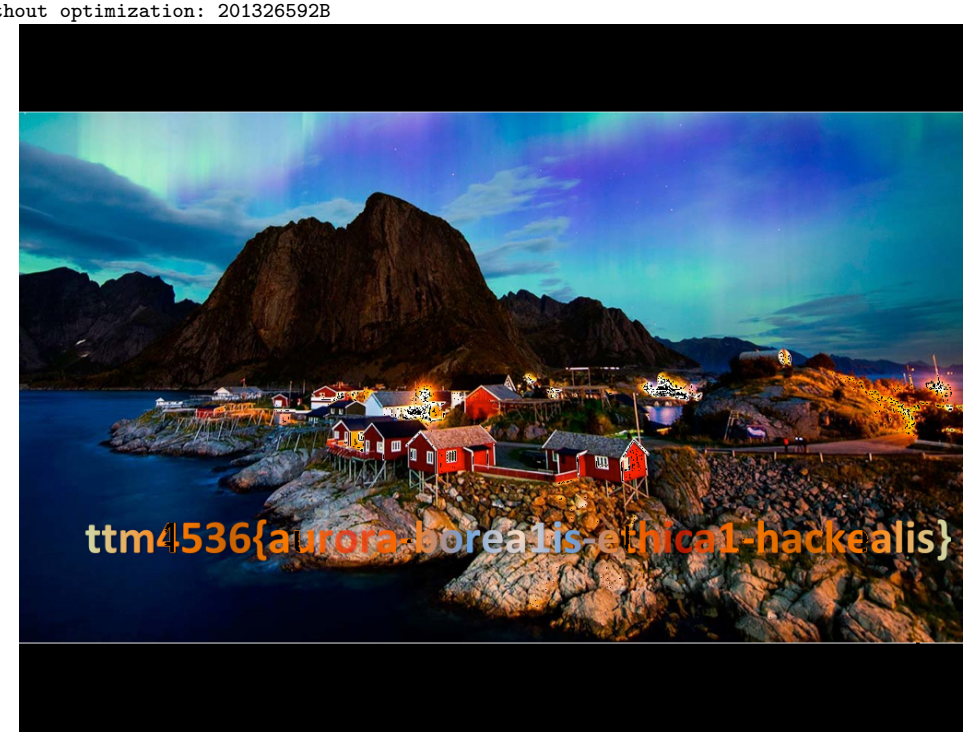
The results of the naïve optimization, even with only halving the pixel range mapped, the result is simply too diffuse to get the full flag, but by trial and error0 with differing step sizes in each channel, it's not unthinkable that one could get the flag like this, with a reduced memory footprint. However, doing so implies a need to tailor the step size for any given image. It's not a generic optimization

We can do better though...

```python
transform_table = {}

def fill_table(img):
    # List all pixels in use for the given image
    pixels_in_use = set()
    for p in [(x, y) for x in range(0, img.size[0]) for y in range(0, img.size[1])]:
        pixels_in_use.add(img.getpixel(p))
    print("number of pixel variations in use: {}, which is {} fewer than the worst case".
          format(len(pixels_in_use), (256**3)-len(pixels_in_use)))

    # Enumerate all pixel variations and test their
    # existence before adding to the lookup-table
    for b in range(0, 255):
        for g in range(0, 255):
            for r in range(0, 255):
                transformed = tuple(transform([r, g, b]))
                if transformed in pixels_in_use:
                    transform_table[transformed] = (r, g, b)

    print("with memory optimization: {}B".format(12*len(transform_table)))
    print("without optimization: {}B".format(12*256**3))
```

Listing 4: Pixel usage optimization of the lookup-reversal; only sample the subset of all pixels that the image we're trying to transform back is using. Drawback: we need to iterate through the entire image first to find which pixel variations are in use.

```
> number of pixel variations in use: 245005, which is 16532211 fewer than the worst case
> with memory optimization: 2925276B
> without optimization: 201326592B
```




The results of the usage optimization, using the transformed image to tailor the selection of which pixels to keep in the lookup-table.

As seen, the optimization drastically reduces the memory footprint, albeit with a temporary increase in usage to list the used pixels in the first place.

- With memory optimization: $2\ 925\ 276_{\mathrm{B}} = 2.79_{\mathrm{MB}}$
- Without memory optimization: $201\ 326\ 592_{\mathrm{B}} = 192_{\mathrm{MB}}$

This concept could be taken even further, given that we know where the flag is located in the image, we could optimize for the specific section of the image in which we expect to find the flag

### 5.2 Brute-force

Out of interest I wondered what would happen if I left the brute-force loop running for longer. At iteration 255 I got my answer:

---


At this point (technically being the 256th iteration considering the original image was one iteration into the transformation already) I reached what I can only assume to be the actual original image before any transformation were applied, as there are no artifacts of transformations whatsoever — just the beautiful northern lights of Lofoten.

The significance of it happening on the 255th iteration, I'm unsure of, as I didn't focus too much on the mathematical approach to solve this challenge, but it seems to not be coincidental that each of the significantly interesting iterations are essentially power-of-twos or factors of power-of-twos minus one.