## Introduction

Steganography is the art and science of hiding information by embedding messages within others. Steganography works by replacing bits of useless or unused data in regular computer files with bits of different, invisible information. This hidden information can be plain text, cipher text, or even images [[\*]](http://www.webopedia.com/TERM/S/steganography.html).

I've implemented two main methods for that purpose: embedText and extractText. You can find the full and documented code with a running example attached to this tip.

## Executing the Code

If you download the attached sample project, it allows you to open an image, write your text or import a text file, optionally encrypt your text before starting processing, embed the text in the image, and finally, save the result image in either PNG or BMP formats. Then you can reopen that image and extract the text from it.

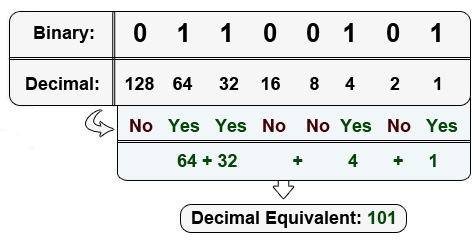


## Explaining the Code

As mentioned before, Steganography works by replacing bits of useless or unused data in regular computer files with bits of our important data. In our case, our data will be the plain text that we need to hide, and the unused data is the least significant bits (LSBs) in the image pixels.

### What is the LSB?

In order to find the decimal value of a binary value, we sum all 1s in the binary value but multiplied by 2n, where n is the position/index of the 1 from the right, starting from zero. For example, to convert 01100101 to decimal, then starting from the right, the equivalent decimal value is 1x20 + 1x22 + 1x25 + 1x26 = 1 + 4 + 32 + 64 = 101



The least significant bit (LSB) is the bit that when flipped from 0 to 1 or from 1 to 0, then no significant change will occur on the total value. It's the bit on the rightmost, that when flipped, the value will be only affected by 1 to be 100 instead of 101. This means that the image will not be significantly affected when we reserve this bit for our purpose. Where the most significant bit (MSB) is bit on the leftmost, that when flipped, the value will be affected by 128 (1x27) to be 229 instead of 101.

When we try to hide our data in an image (as in our case), then we need enough budget of LSBs to hide our data in. These bits are located in the image pixels. Since each pixel has three elements (R, G, and B that represent the Red, Green, and Blue elements of the pixel consecutively, assuming non-transparent image), each of these elements can have a value between 0 and 255. Now, assume that the image was 300 pixels width by 400 pixels height, then we'll have 300 x 400 x 3 = 360000 LSBs. And as each character can be represented by 8 bits, then that image can hide 360000 / 8 = 45000 characters!

#### 

#### 

#### **Step by Step**

1. **Hiding the text inside the image**
   * Loop through the pixels of the image. In each iteration, get the RGB values separated each in a separate integer.
   * For each of R, G, and B, make the LSB equals to 0. These bits will be used in hiding characters.
   * Get the current character and convert it to integer. Then hide its 8 bits in R1, G1, B1, R2, G2, B2, R3, G3, where the numbers refer to the numbers of the pixels. In each LSB of these elements (from R1 to G3), hide the bits of the character consecutively.
   * When the 8 bits of the character are processed, jump to the next character, and repeat the process until the whole text is processed.
   * The text can be hidden in a small part of the image according to the length of that text. So, there must be something to indicate that here we reached the end of the text. The indicator is simply 8 consecutive zeros. This will be needed when extracting the text from the image.
2. **Extracting the text from the image**It's more simple than hiding. Just pass through the pixels of the image until you find 8 consecutive zeros. As you are passing, pick the LSB from each pixel element (R, G, B) and attach it into an empty value. When the 8 bits of this value are done, convert it back to character, then add that character to the result text you are seeking.