Second International Conference
Modelling and Development of Intelligent Systems
Sibiu - Romania, September 29 - October 02, 2011

"Least Significant Bit" method in steganography

Gabriel TUDORICĂ, Paul STÂNEA, Daniel HUNYADI

Abstract

The purpose of this paper is to present the benefits of steganography and emphasize popular ways it can be applied to. We will focus on a digital steganography technique, using Bitmap files as carrier files for our hidden messages, thus hiding it in plain sight. Even though the picture can be seen by others, only the sender and the intended recipient will actually be able to get the messages.

Finally, we built an application using C# capable of applying this steganography technique, and allowing the user to embed hidden messages in 24 bit Bitmap files. Additionally, we implemented a chat-like environment using 24 bit Bitmap files to send the hidden data.

1. Introduction

We built our steganography application on the .NET Framework, using C# and it was designed to allow any user to embed hidden messages into 24 bit Bitmap files.

Using a substitution method called Least Significant Bit, this feature was made possible. This substitution method has a drawback though; namely the amount of text that can be hidden is limited to the size of the bitmap.

Basically, our application loads any 24 bit bitmap and using the Least Significant Bit method, it calculates the maximum number of bits (in characters) one can embed into the file. The substitution method replaces the last bit of every byte in the bitmap file with the bits from our binary transformed message. Clearly, our message must not exceed the maximum number of bits available for this process. Thus, we built a limiter which provides graphical feedback to the user.

After the message has been entered, for security reasons, the user must enter a password to protect the file for the decoding process.

After the password has been set, the application saves the new bitmap file with the embedded message and password and the new file can be sent to the intended recipient. The differences between the original file and the one with the embedded message are so small, that no one will notice unless they can compare it to the original.

In most cases, depending on the bitmap size and hidden message length, there were no visual and/or size differences between the two.

For the decoding part, the recipient will load the file into the application, fill in the password field, and the hidden message will be revealed.

Since there are other applications capable of doing the same thing, we thought of a way to significantly improve ours. While most steganography applications use one technique to embed messages, we implemented another one, namely the generation technique.

Two out of three techniques have some serious drawbacks!

More precisely, the insertion method adds harmless bits to an executable file for example, increasing the carrier file size depending on the embedded message length, thus arousing suspicion.

The big drawback of the substitution method is that the maximum available size for our hidden message is determined by the size of our bitmap file, which can cause problems.

Since the two above mentioned methods are not 100% trustworthy, we used a third one, the generation method, which generates a large enough image to carry the entered message. This method requires the message to be entered first, then it calculates the size of the text in bits and finally generates a large enough 24 bit bitmap file capable of embedding the message. The resulting bitmap file size will be directly dependent to the message length and its width and height are chosen so that the resulting file is large enough to withhold the message and not too much larger then it is needed. Since this is directly linked to the message length, we won't have 2 images of the same width or height.

Furthermore, we added another great feature to our application using the generation method. We created a chat-like environment where two users can chat using secure hidden messages found in 24 bit bitmap files using a FTP server. Once the messages are created, a picture carrying the message will be generated and uploaded to a FTP server, where the other instance of the application scans the server for new images.

When found, it downloads it locally, loads it, decodes it and displays the message. After the message is displayed, both the remote and local files are deleted. We did not protect the images with a password because we wanted to speed up the whole process and make it as close as a chat-like application as we could. And since the carrier files will stay for just about a few seconds on the server, the file does not present a security risk.

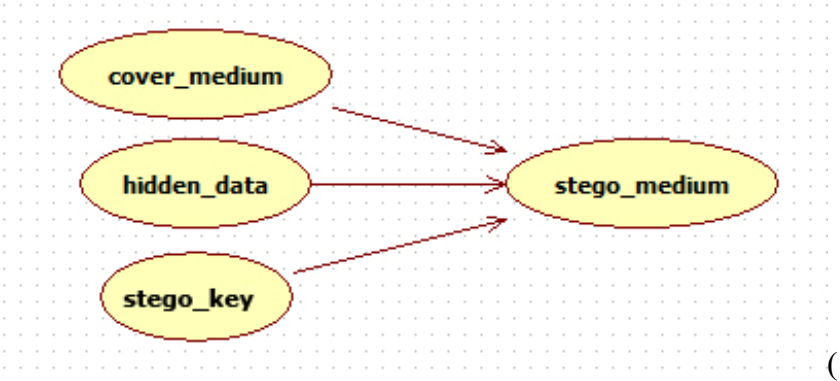
The main goal we managed to achieve here is that the whole conversation, viewed from outside, is camouflaged into a trivial file upload on a FTP Server, which is great!

2. Steganography

Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and the intended recipient, suspects the existence of the message. It's a form of security through obscurity.

The main advantage of using steganography over cryptography alone, is that the messages do not attract attention on themselves, furthermore the messages are double protected. [1][2][3]

Diagram (1) provides a very generic description of the pieces of the steganographic process:



(1)

Cover_medium is the file in which we will hide the hidden_data (the carrier file), which may also be encrypted using the stego_key. The resultant file is the stego_medium, which will be the same file type as the cover_medium and it will be perfectly usable. The cover_medium are typically image or sound files. Our application will focus on image files.[2][3]

History confirms what we already know: the best place to hide messages is in plain sight. An example of ancient steganography use is that of Histiaeus, who shaved the head of his most trusted slave and tattooed a message on it. After his hair had grown, the message was hidden. His purpose was to revolt against the Persians. [1][2]

Another useful purpose of steganography is the so-called *digital watermarking*. A watermark, historically, is the replication of an image, logo, or text on paper stock so that the source of the document can be at least partially authenticated. A digital watermark can accomplish the same function.

A graphic artist, for example, might post on a website sample images with an embedded signature (known only by him), so that he can later prove his ownership in case others attempt to portray his work as their own. [3]

2.1 Techniques

2.2 Physical steganography

Steganography has been and it is used today in many forms. The ways of hiding data nowadays are endless, but the most popular include:[4][5]

- Messages tattooed/written on messenger's body;
- Messages written in secret inks;
- Messages written below the postage stamps on envelopes.

2.2.1 Digital steganography

Steganography evolved in 1985 with the advent of the personal computer. Since then, there have been built over 800 steganography applications recognized by official institutions around the world. Some common digital steganography methods are: [4][5]

- Hiding messages in the lowest bits of noisy images or sound files;
- Hiding data in encrypted data or within random data;
- Adding harmless bits to executable files;
- Embedding photos in video material;

2.2.2 Network steganography

Network steganography uses communication protocols' control elements and their basic functionality, making the whole process even harder to detect or remove. An example of network steganography method is **Steganophony** - the concealment of messages in Voice-over-IP conversations, e.g. intentionally sending corrupted packets that the receiver would ignore by default; [4][5]

2.2.3 Printed steganography

Nowadays much of the steganography employed today is quite high-tech, but steganography's goal is to mask the existence of a message. A message can be concealed by traditional means and produce a ciphertext.

A popular and almost obvious method is called a null cipher. In this type of steganography, one would decode the hidden message by taking the first or other fixed letter from each word and create new words. Other forms include deliberately making mistakes, using different fonts or other hard to notice symbols. [4][5]

Consider this cablegram that might have been sent by a journalist/spy from the U.S. to Europe during World War I:

P*RESIDENT'S* ***E****MBARGO* ***R****ULING* ***S****HOULD* ***H****AVE* ***I****MMEDIATE* ***N****OTICE.* ***G****RAVE* ***S****ITUATION* ***A****FFECTING* ***I****NTERNATIONAL* ***L****AW.* ***S****TATEMENT* ***F****ORESHADOWS* ***R****UIN* ***O****F* ***M****ANY* ***N****EUTRALS.* ***Y****ELLOW* ***J****OURNALS* ***U****NIFYING* ***N****ATIONAL* ***E****XCITEMENT* ***I****MMENSELY.*

The first letters of each word form the character string: *PERSHINGSAILSFROMNYJUNEI*. A little imagination and some spaces yields the real message: *PERSHING SAILS FROM NY JUNE I.*

3. Algorithms and Techniques

There are three different techniques you can use to hide information in a cover file:

a) Injection (or insertion);
b) Substitution
c) Generation

3.1 Injection or insertion

Using this technique, the data is stored in section of a file that is ignored by the application that processes it. For example in unused header bits or adding harmless bytes to a file leaving it perfectly usable.

The more data you add, the larger the file gets, and this can raise suspicion. This is the main drawback of this method.[1][4]

3.2 Substitution

Using this approach, the least significant bits of information are replaced with desired data, which will be reproduced on decoding.

The main advantage of this method is that the file size does not change, but the file can be affected by quality degradation, which in some cases may corrupt files. Another flaw is that the available amount of data is limited by the number of insignificant bits in the cover file. [1][4]

3.3 Generation

Unlike injection and substitution, this technique doesn't require an existing file, it generates it just to embed the message, which is better than the other two mentioned methods because it’s not being attached to another file to suspiciously increase the file size, it has no limitation and the result is an original file and therefore immune to comparison tests. [1][4]

4. The Least Significant Bit (LSB) method

The Least Significant Bit (LSB) method is the most common substitution technique, which basically replaces the least significant bytes in a file to conceal data, which is extracted at the decoding process. It's especially effective on lossless image files, such as 24 bit Bitmap files.

The method takes the binary representation of any form of data and overwrites the last bit of every byte in an image.

As an example, we will take a 3x1 pixel 24 bit bitmap, each pixel color is represented on one byte so our picture bitmap will have a total of 9 bytes. We have the following RGB encoding:

11010101 10001101 01001001
11010110 10001111 01001010
11011111 10010000 01001011

Now suppose we want hide the following 9 bits of data: 101101101. If we overlay these 9 bits over the original file using the LSB method, we end up with the following new RGB encoding, which visually is not too different. The bits in **bold** have been changed.

*11010101 1000110**0** 01001001*
*1101011**1** 1000111**0** 0100101**1***
11011111 10010000 01001011

We managed to hide 9 bits of information by only changing 4 bits or about 50% of the lest significant bits.[2]

Similar methods can be applied to lower color depth image files, but the changes would be too dramatic and therefore obvious. On the other hand, grayscale images provide an excellent cover file for the LSB method.

It is worth mentioning that *steganalysis* is the art of detecting and breaking steganography. A form o analysis is to carefully examine the color palette of an image, which very often has a unique binary encoding for each individual color. If steganography is used, the same color will end up having more binary encodings, which would suggest the use of steganography in the specific file.

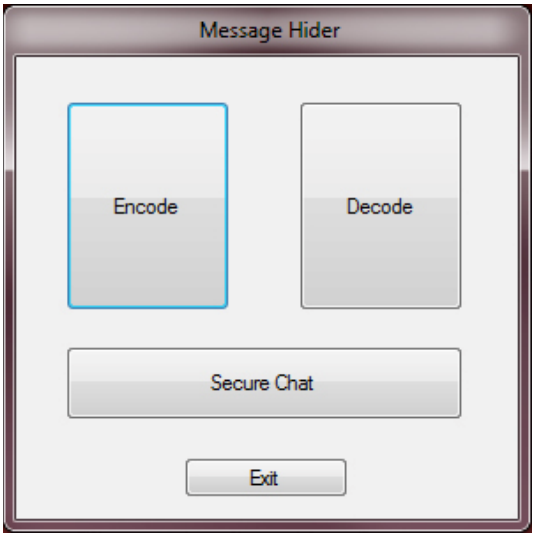
But if we wanted to check for hidden information in images, which files should we analyze? Suppose I decide to send a message hidden in a picture that I will post on a website, where others can see it, but only certain people know that it contains hidden information. I can also post more images and only some of them would have hidden data so the potential steganalyst would get confused. [2][5]

The quantity of potential cover files make steganalysis a Herculean task, and we will exploit this very spot of steganalysis in our application.

5. MessageHider Application

MessageHider is a steganography application we developed using .NET Framework and C#. Compared with other software products that use only one steganography technique, our application uses two: substitution and generation. Our application has two modes: normal and chat mode. In the following, we will present the application in detail.

The Graphic Interface is basic, and allows the user to enter the two modes: normal and chat mode.



Main screen (2)

5.1 Normal mode

The normal mode, provides the basic functionality, using a substitution algorithm, more precisely the Least Significant Bit method. We will use 24 bit bitmaps as stego files, because of their lossless compression. Normal mode provides features for encoding and decoding hidden messages.

5.1.1 Encoding

To encode a hidden message, the user needs to first select the source file (24 bit bitmap). Once the bitmap is loaded, our application will estimate the maximum number of characters the user can hide in the selected file. As we have seen, the Least Significant Bit method, replaces the last bit of every byte in the file, with the bits for our hidden message. To get the maximum number of characters a user can hide we will use the following formula (ImageWith, ImageHeight are expressed in pixels):

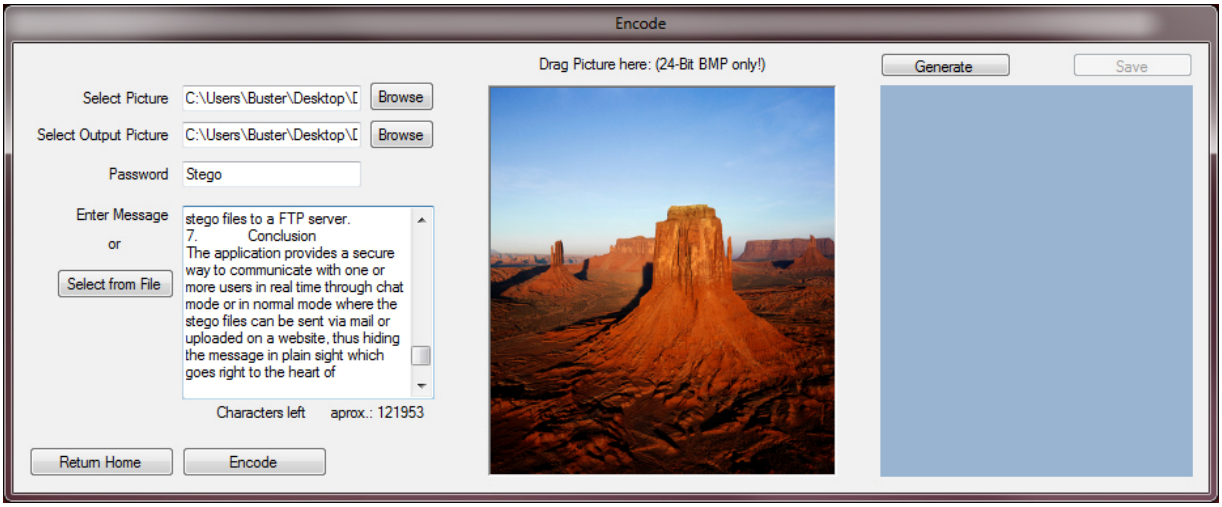
$$\text{AvailableCharacters} \approx \text{CEILING}\left(\frac{1}{16}\left(\frac{\text{ImageWidth}\bullet\text{BitsPerPixel}}{8}\,\text{ImageHeight} + \text{BmpHeader}\right)\right) \quad (3)$$

In other words, for a 1024x768 bmp image we will have 147459 characters.

$$\text{AvailableCharacters} \approx \text{CEILING}\left(\frac{1}{16}\left(\frac{50 \bullet 24}{8} 50 + 54\right)\right) \approx 473 \text{ characters} \tag{4}$$

When a bitmap is being loaded, the user will have a visual feedback for the maximum characters available as the message is being typed and will also limit the maximum number of characters he enters to the maximum available characters calculated before minus 32. The 32 characters we subtracted are reserved for the password, which can be by up to 30 characters, the two characters left being used as a delimiter from the original message on which the password will be concatenated.

After the message was entered and the password set, the Least Significant method will be called and the stego file will be generated.



Encoding 1024x768 bmp (5)

The form the user will have to fill in has the following fields:

- Select picture – the path to the carrier image in which we will hide our message. The image will be a 24 bit bitmap;
- Select output picture – the path to the output picture;
- Password – the password required for the decoding of the message;
- Message – the message we want to hide;

We also provided a drag and drop box so that the user can drag the bitmap in the box and the form will autocomplete its path. We can also load text from a file to speed up the process.

We actually managed to hide our entire article (without the images) in a single 1024x768 standard Windows 7 image which we converted to BMP format. Our article used only 18% of the available characters we could hide in the picture, having another 121,953 characters left.

We would also like to point out the fact that the size of the image did not change, both the orginal and the stego file had 2,359,352 bytes. Also, from a visual perspective, both files are identical.

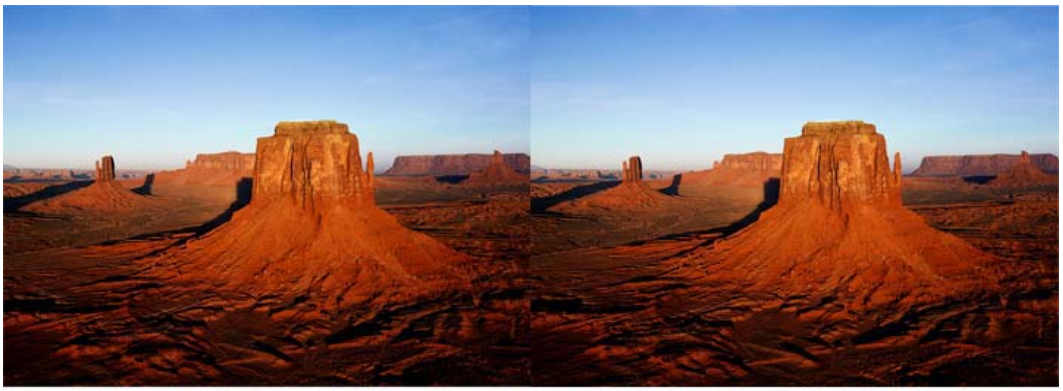
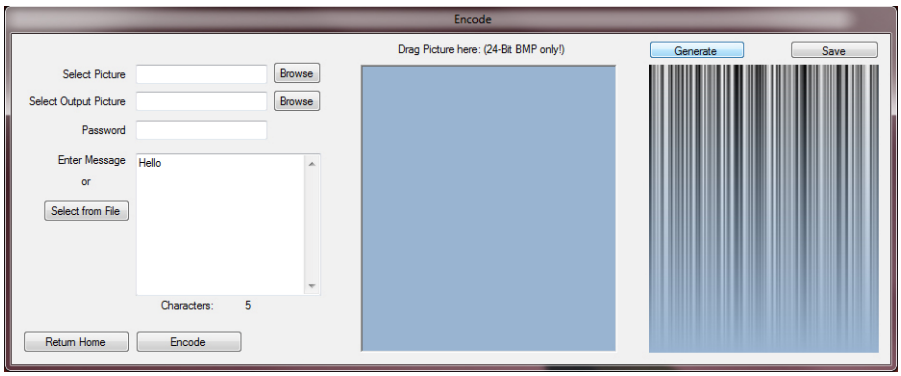


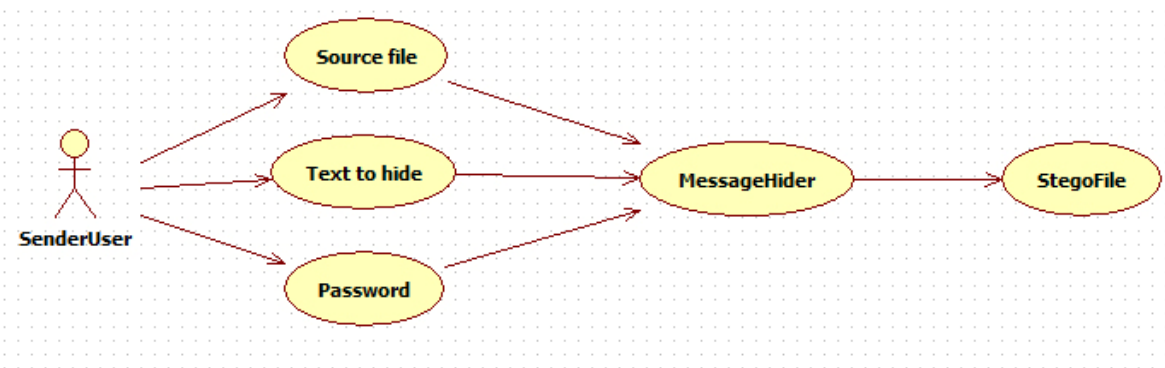
Image comparison (6)

Last, but not least we added a bitmap generation module so that the user could generate a large enough bitmap to withhold the entered message. The resulting bitmap will be black and white with lots of noise.



Encoding with generated b&w noisy bitmap (7)

The whole process can be visualized using diagram (8):



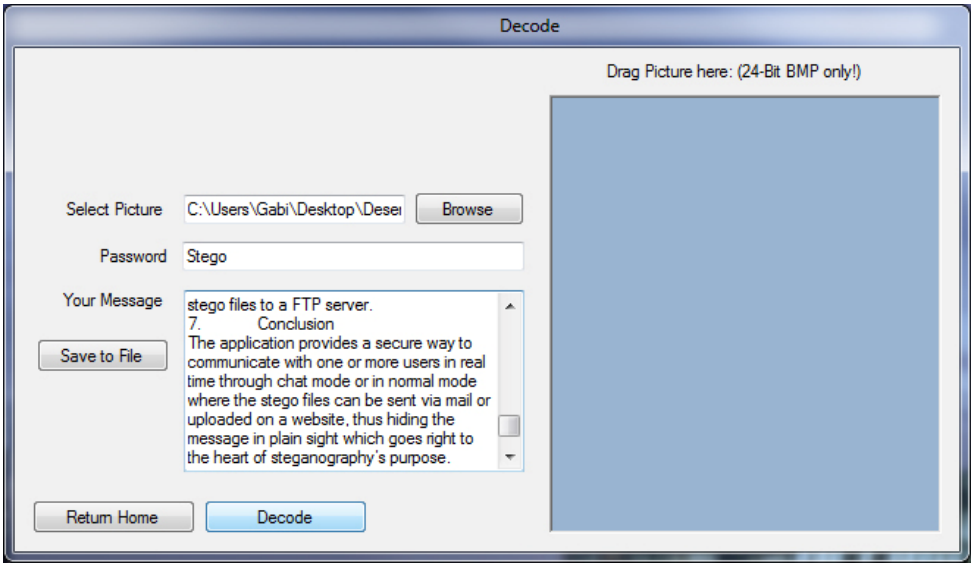
(8)

We can conclude that diagram (8), is the extended version of the diagram (1) we presented earlier.

5.1.2 Decoding

Firstly, the user will load the stego file into the application. Once this is done, the Least Significant Bit method will extract the last bit from every byte of the stego file and thus recreate the original binary message. After a conversion from byte to string, we get the original message. The message also has the password concatenated and delimited by the || characters.

The password is extracted from the string and stored for it to be checked in the next step, where the user is required to fill in the password field. If the password is correct, the original message sent will be revealed to the user.

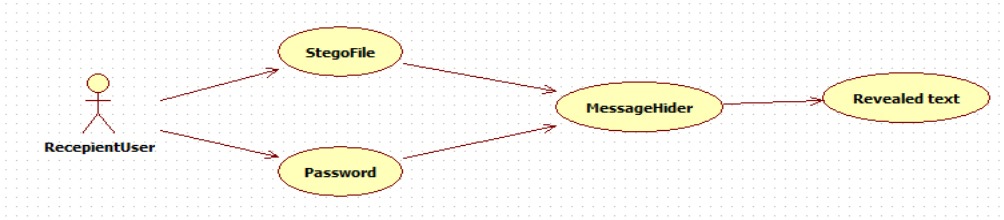


Decoding (9)

The decoding form has the following fields:

- Select picture – the path to the bitmap picture we want to extract the message from, which, as before, it can be obtained if the user drags and drops the file in the drop box;
- Password – the password we protected our message with;
- Your message – the area where the original message will be revealed if the password is correct.

The whole process can be visualized using diagram (10):



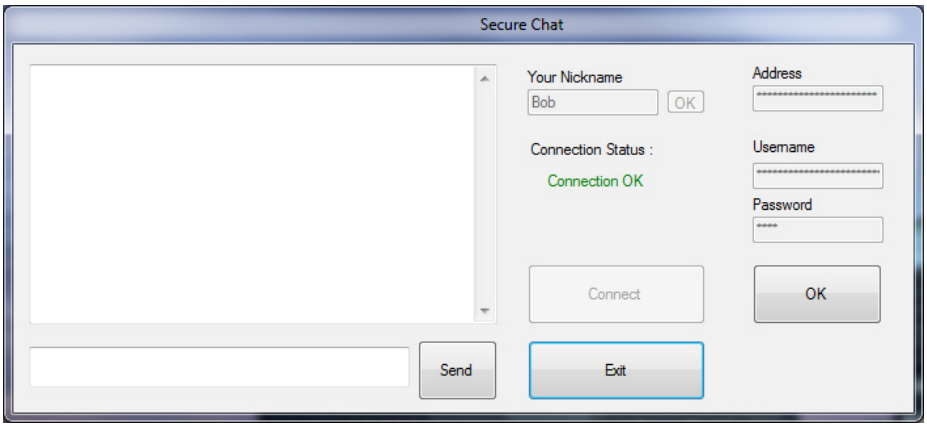
(10)

As diagram (10) shows, the decoding process is the inverse of the encoding process.

5.2 Chat mode

Chat mode uses the generation steganography technique. Basically, the user first types the message he wishes to hide and a large enough bitmap capable of embedding the message will be generated as a stegofile. The message will be protected by a generic password. Once the message was embedded, the stegofile will be named using a convention based on the nickname the user chose (which he previously filled in) and a timestamp. Then, the stegofile will be uploaded to an FTP server, using the credentials filled in by the user from the form, and the message he wrote will be displayed in the textfield.

The other user will connect on the same FTP server and the application will scan for the existing files that do not start with the current user's nickname based on the convention we talked about earlier. If found, the bitmaps will be downloaded from the FTP server, decoded using the same generic password and the message will be displayed in the textfield along with the other user's nickname. The local bitmaps and the ones on the FTP server will be erased as soon as the the bitamp is decoded.



Secure chat mode (11)

The main advantage is that the whole chat can be seen as a trivial uploading and deleting procedures.

Once in chat mode, the user will have to first assign a nickname which will be used to differentiate the message sender. Once the user fills in the nickname field, he will fill in the credentials in order to connect to the FTP Server and lastly press the Connect button. A visual feedback will notify the user about the connection status.

Once connected to the FTP Server, the user can begin to exchange messages with the other user. Using the message textbox, the user can type in the message he wishes to send. When he presses the send button, a couple of processes are initiating.

Firstly, we need to generate a picture large enough for our message to fit in. The width of the image will be generated by a random number between 100-300 pixels, all we have to calculate now is the height. Based on (3), we conclude that the ImageHeight is:

$$\text{ImageHeight} \approx \text{CEILING}\left(128 \frac{\text{MessageLength}}{\text{ImageWidth} \cdot \text{BitsPerPixel}}\right) \quad (12)$$

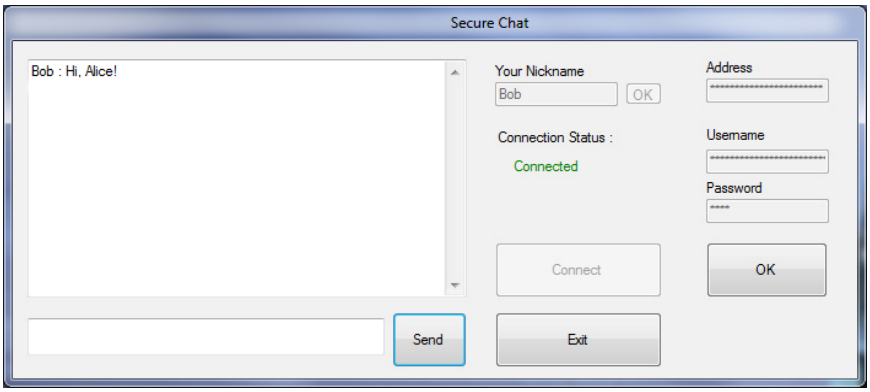
Now that we have the ImageWidth and ImageHeight, we can generate the carrier image. We just added noise to the image in order not to leave it blank, but random graphics can be generated or even fractals. Actually we have this in mind for future developments.

Once the image is generated, we apply the Least Significant Bit method, previously described in the normal mode – encoding part. After the message is embedded, we upload the bitmap file to the FTP Server and rename it as [nickname--unixTimestamp.bmp].

By using this naming convention we can easily determine the message sender, and sort the files/messages in chronological order because of the unixTimestamp. Once the image has been uploaded, it is deleted from the local path.

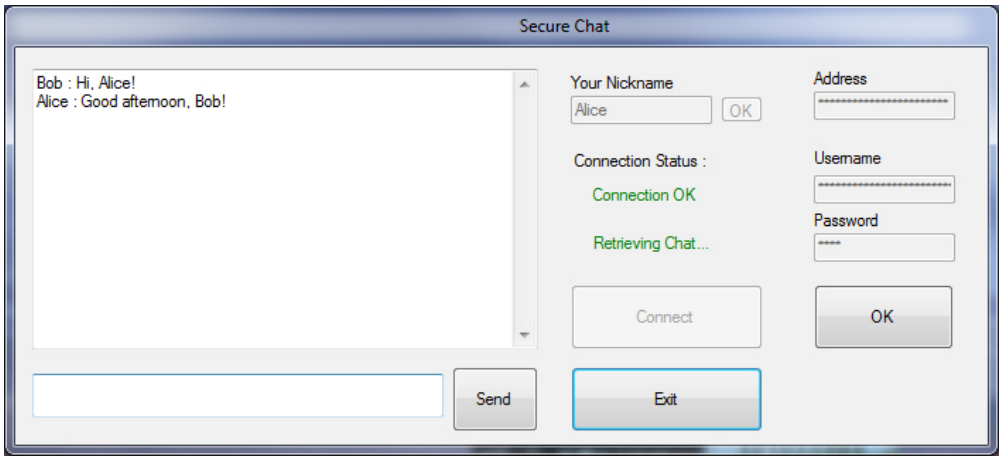
Bob--1316905935.bmp – zoomed in 291x1 pixel bitmap generated for the Hi, Alice! message sent by Bob (13)

The message we embedded in the file will also be displayed in the chat window, having the user's nickname as a prefix.



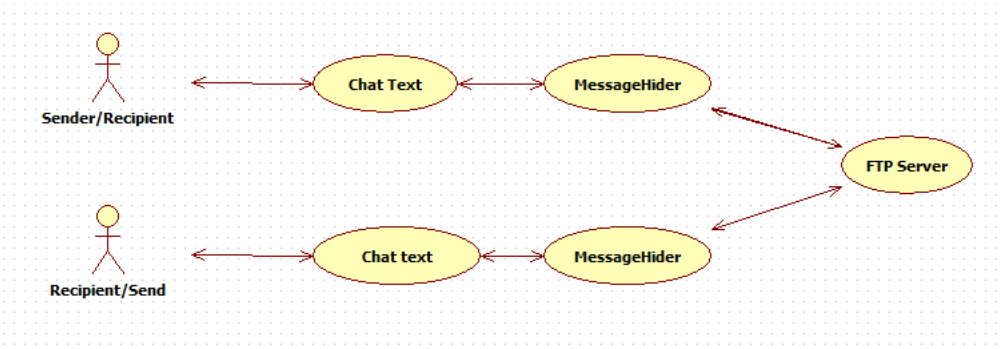
Secure chat mode – Bob (14)

Once the file is decoded, both the remote and local files are deleted. When the user closes the chat mode, the FTP connection will close as well.



Secure chat mode – Alice (15)

Visually the whole process can be visualized using diagram (16):



(16)

As diagram (16) shows, our application in chat mode is similar to a native chat application, only instead of using sockets and a chat server, our application uploads stego files to a FTP server.

6. Conclusion

The application provides a secure way to communicate with one or more users in real time through chat mode or in normal mode where the stego files can be sent via mail or uploaded on a website, thus hiding the message in plain sight which goes right to the heart of steganography's purpose.

Of course, there is always room for improvements. For future implementations we thought of encrypting the message using strong cryptography algorithms and embed the ciphertext in the bmp files, thus creating a double protection. Another feature we thought of, was to implement methods capable of embedding the messages in other file types (other image types or even sound or video files).

7. References

[1] http://en.wikipedia.org/wiki/Steganography
[2] http://www.garykessler.net/library/steganography.html
[3] http://www.garykessler.net/library/fsc_stego.html
[4] http://www.infosyssec.com/infosyssec/Steganography/techniques.htm
[5] http://en.wikipedia.org/wiki/Least_significant_bit
[6] http://en.wikipedia.org/wiki/BMP_file_format
[7] http://www.fastgraph.com/help/bmp_header_format.html

Gabriel TUDORICĂ
"Lucian Blaga" University of Sibiu
Faculty of Sciences
Sibiu, Dr. Ioan Ra iu St. No. 5 - 7
ROMÂNIA
E-mail: office@eyenetworks.ro

Paul STÂNEA
"Lucian Blaga" University of Sibiu
Faculty of Sciences
Sibiu, Dr. Ioan Ra iu St. No. 5 - 7
ROMÂNIA
E-mail: psb77black@yahoo.com

Daniel HUNYADI
"Lucian Blaga" University of Sibiu
Faculty of Sciences
Sibiu, Dr. Ioan Ra iu St. No. 5 - 7
ROMÂNIA
E-mail: daniel.hunyadi@ulbsibiu.ro