Using digital images to spread executable code on internet

Roberto Gómez , Gabriel Ramírez[1]

[1] ITESM-CEM, Depto. Ciencias Computacionales, Km 3.5 Lago Guadalupe, 51296, Atizapan Zaragoza, Edo México, Mexico
{rogomez, A00472181}@itesm.mx

Abstract. . Steganography is defined as covered writing and it has been used to achieve confidentiality. Most of research in this domain has focused in just sending plain data. In this paper we use steganography as a way to spread executable code instead of just communicating a secret message. The application of our proposal may contrast, could be using for spreading malicious code or for protecting software copyright like a water marking alive. We present our implementation of this scheme with a developed tool and hence look in a direction that has not looked at detail so far.

1 Introduction

Steganography has evolved during history from handcrafted ways to very complex techniques, giving to the humans a way to conceal communication. However our purpose is to use steganography as a mechanism to hide executable code, using the power of steganography not to conceal communication, but to conceal the transportation of executable code.

Internet was a major breakthrough in the way we communicate with each other bringing a universe of information. This universe has a proliferation of digital images, being the most interchangeable kind of file. Multimedia data presents a highly redundant representation which usually allows the hide of significantly large amounts of data. Due to this, image files are the ideal object to hide information, especially executable code, besides other kind of information.

Most of the remote code execution schemes in Internet are based on a client/server model. A server listens on specific port, and the client sends a request to that port. Once the request arrives, and depending in the type of request, the server performs a specific action. In our model, a server waits for the arrival of an image, once it arrives, the server looks for the code embedded into the image file. If any code is found in the image, our system extracts the code, load it and execute it. If not code is detected it continues searching for other image files.

As far as we know there is not enough research about using steganography to propagate executable code. We propose a protocol to propagate code using the principles of steganography.

This paper is organized as follows: in section 2 we introduce ht main features of steganography. Section 3 presents BMP image file format. In section 4 we explain related work of steganography with executable code and the differences with our own approach. In section 5 we explain our model. In section 6 we show our experiments and results. In section 7 we give our conclusions, and the future work.

2 Steganography

The term steganography is derived from the Greek words steganos which means "covered" and graphia which means "writing", i.e. covered writing. Steganography refers to the art and science of concealing a communication; unlike cryptography, where conceals the message but the communication is often known [1]. In fact the ideal would be a combination of these areas to accomplish a framework more robust to assure secure communications. Steganography is considered as an art because it is related with creativity and it is also considered as a science due to the areas involved. These two approaches combined give a powerful area in computer science. Classical examples of steganography are invisible ink and microdot.

The steganographic process involves the following elements: a carrier, a secret message, a key(s), an embedding algorithm and a protocol. In digital steganography, a carrier may be any kind of digital file for example a picture, movie, audio, etc; as well as the secret message. The key or keys could be any combination of steganography and cryptography keys and they may feed the embedding algorithm in order to generate certain seed to incrust the secret message based on that input. Obviously like in every kind of communication is needed a protocol, in this case is very similar what is managed in private and public key cryptography or could be without the interchange of any key, what is known as pure steganography. All these elements are used in order to generate an object which is known as a steganogram, the carrier with the secret message embedded. The perceptual content of the cover object must be indistinguishable from the setganogram in order to consider the steganographic process effective and efficient.

Basically there are three ways to hide information into a digital object: adding the secret message, generating a steganogram from scratch (cover generation), and substituting data from the carrier. In [2] we found a good survey of these techniques.

The adding techniques incrust the secret message into a carrier without modified data at all. The technique appends the secret information in certain parts that do not affect the quality of the carrier itself. The disadvantage is that the size of the resultant steganogram files increments as much as the size of the secret message, looking suspicious if the size of the secret message is significant. Nevertheless the technique does not generate any degradation on the carrier. Examples of these techniques consist in hide information in not used fields of TCP/IP headers [3], or to hide it after the end of a word file.

The cover generation techniques encode information in the way a cover is generated. They have the advantage that the generated steganogram presents certain properties to defend it against statistical attacks. By the other hand the resultant steganogram might not be common and look suspicious. As an example we can mention

the Automated Generation of English Text, which uses a large dictionary of words categorised by different types, and a style source which describes how words of different types can be used to form a meaningful sentence. It transforms the message bits into sentences by selecting words out of the dictionary which conforms to a sentence structure given in the style source. Another example is a fractal image.

The substitution techniques are the most used by the steganographic tools. They substitute redundant parts of a carrier with a secret message. One of the simplest substitution methods is LSB (Least Significant Bits); it chooses a subset of cover elements and substitute least significant bit(s) of each element by message bit(s). LSB take advantage that the human eye is not capable to distinguish slight modifications in the carrier. The receptor of the message could only extract the message if he knows the exact positions where it was incrusted. This technique has been used in image, audio and video files, taking advantage of redundancy. It can be used en grey- scale and colour images with one, two, three or even four least significant bits. The disadvantage of this technique is that the information hidden can be altered or erased if the carrier is compressed or filtered by any method. An excellent study of LSB technique can be found in [4].

A pseudorandom number generator may be used to spread the secret message over the cover in a random manner. It must be noted that, according to [5], that classical definition of steganography is statistical and not perceptual. In his paper Cachin defines a steganography technique to be E-secure if the relative entropy of the probability distribution of cover objects and steganogram is less than or equal to E. He calls a steganography technique to be perfectly secure if E is zero. He then demonstrates such steganographic techniques do exist and they are perfectly secure (however, the technique described by him is impractical).

A different approach in substitution is image downgrading. It differs from the previous one in the quality of the final image. An image is said to have been downgraged when its sensitivity label has been changed to the one strictly dominated by its previous value, [6] For example a representation of 32 bits is reduced to 16 bits, with a quality degradation.

According to [7] there are two kinds of steganographics attacks, visual and statical attacks. More reliable attacks have been published for sequential (or not) LSB embedding and can be found in [8].

Westfeld and Pfitzmann establishes that the idea of visual attacks is to remove all parts of the image covering the message, so the human eye can distinguish whether there is a potential message or still image content. The filtering process depends on the presumed steganographic method used. So the attacker needs to known the used technique.

By the other hand the idea of the statistical attack is to compare the theoretically expected frequency distribution in steganograms with some sample distribution observed in the possibly changed carrier medium. The attacker needs to compare the original carrier with the steganogram in order to detect this. In order to do this the attacker needs the original carrier, and this does not occur in most of the cases.

Both techniques do not allow the attacker to obtain the hiding data, but it gives enough information to detect that something is embedded in the carrier.

Another substitution technique that defends to statistical attacks is pseudorandom permutations; it takes random bits and bytes from carrier and replaces them with the message performing some mathematical computation. Using this technique provides more robustness against statistical analysis.

There are more robust substitution techniques, known as transformation techniques [9]. These techniques incrust the secret message in significant data of the carrier based on a mathematical process that take the signal from one representation to another one, surviving only the significant data. The disadvantage of this method is the difficulty to find places where to hide the secret message, because redundancy is reduced at minimum.

Our proposal is based in substitution techniques and specifically LSB due to its simplicity, and efficient in implementation. The size of the resultant steganogram does not change at all, unless the carrier can not afford space enough for transporting the executable code. Besides the degradation of the steganogram does not look suspicious and it can not being used to detect that the carrier hides some information.

3 The BMP file format

Digital image formats are ways to represent images in digital form. The image is represented as a finite set of digital values, called picture elements or pixels. There are many different digital image formats in use today, but certainly the most used today are: BMP, GIF and JPEG.

The basic elements of a image file format are: fields, labels and blocks. Mastering these components is crucial to know where and how hide data into them.

GIF images are used extensively on the web. Supports animated images. The format supports only 255 colors per frame, so it losses quantization in full-color photos. It is based on a color table with a maximum of 256 colors. This format is popular for its reduced size based on LZW compression and because it allows small animations and interlacing.

JPEG divides an image in blocks of 8x8 pixels and applies a mathematical transformation to keep only the significant data and get rid of the data that is not important to visualize the image. By doing this, the image size is smaller. This image format is ideal to store pictures because of its powerful compression based on a DCT coefficients.

It is important to remark that JPEG itself specifies only how an image is transformed into a stream of bytes, but not how those bytes are encapsulated in any particular storage medium. A further standard, created by the Independent JPEG Group, called JFIF (JPEG File Interchange Format) specifies how to produce a file suitable for computer storage and transmission (such as over the Internet) from a JPEG stream. In common usage, when one speaks of a "JPEG file" one generally means a JFIF file.

BMP image files are commonly used by Microsoft Windows programs, and the Windows operating system itself. The BMP format consists basically on the following structures: a file header, an image header, a color palette and the data (pixels). According to the MSDN library, all the structures specify three main elements. The

first element is the header that describes the resolution of the device on which the rectangle of pixels was created, the dimensions of the rectangle, the size of the array of bits, and so on. The second element is a logical palette. And the third element consists of an array of bits that defines the relationship between pixels in the bitmapped image and entries in the logical palette.

The file header contains the first 14 bytes of the file, and it contains information about the bitmap data found elsewhere in the file. Next to the file header there is the image header. It has a length of 40 bytes, and it holds the fields that describe the image itself. Bitmaps are usually organized, either physically or logically, into lines of pixels. Following the image header we found the number of elements in the color table. This number is equal to the number of colors in the image. It is not common using a color table. In the 24-bit uncompressed RGB color mode, there is not a color table, each pixel consists of three 8-bit values, a blue byte, a green byte, and a red byte. The combination of the three values represents the pixel's color, and they represents all the image pixels lineal, from left to right upwards line-by-line starting at the lower left. This format conformation gives a high degree of redundancy and is possible to substitute the least significant bits of the carrier.

In addition, it could be used a fourth reserved byte of every pixel representation in the case if it exists a color table.

We choose to work with BMP format due to its plain structure and the extremely redundancy used to represent every single color of one pixel in digital images. An observer will not be able to distinguish between a carrier and a steganogram in a BMP file due to eye capacity to distinguish that kind of slight changes.

Implementing LSB in GIF and JPEG formats would imply decoding data flow first according to the compression process and then applied LSB to the data to finally decode for a second time with the compression process again.

4 Related work

Our idea is based in the techniques used in steganography to hide information, but instead of hiding data, we propose to hide executable code. This code will be extracted and executed once it arrives to a computer. As far as we know there is no substantial research in this kind of steganography. We found some empirical work of this approach

In the master degree thesis presented in [10] the author exposes the possibility of an attack in computer warfare. The author installs a steganographic parser in a computer target using an executable wrapper in form of video game, like a Trojan horse, which needs a previous intervention of the computer affected user. The author embeds executable code into a different format files like BMP, GIF, JPEG, TXT, and HTML, employing public steganographic tools available like S-Tools. Once the parser steganographic is installed every steganographic file may be used by this one to extract and execute code.

Another similar approach is presented in [11]. The authors employ an ActiveX installer in a Web page to install the steganographic parser. This approach works with the comment label of the JFIF format, a popular implementation of the JPEG stan-

dard. The authors embed a keylogger, which is a kind of spyware in the system that records every single action in the keyboard. The authors propose that, after certain time, the keylogger send the logs (with valuable information) to the perpetrator of this attack.

The use of an AVI file to hide executable code is proposed in [12]. The authors take advantage that this kind of file launches an event, so it is not necessary a previous installation of a steganographic parser. They propose to force the event in such a way that it runs out of a web browser, executing the hiding code without the need of the user authorization.

A slight different approach that takes advantage of a specific vulnerability in Microsoft software is presented in [13]. It consists of a buffer overflow, "activated" when an image JFIF format file is created with a comment block with one or zero bytes of size. Knowing that any block in JFIF format consists at minimum of two bytes because includes the two bytes that indicate the size of the bock, Microsoft libraries does not know how to deal with this exception and might produce buffer overflow in the system.

In [14] the authors give a proof of concept that demonstrates propagating a computational virus using steganography. The author appends the executable code after the EOF label in a JFIF picture. It needs a previous intervention of the user to install the steganographic parser and infecting the first JFIF image file in the system. This approach spreads the executable code in a picture for every image infected being visualized in the same path.

5 Our proposal

The main objective of our system is to spread executable code embedded in BMP images. In order to accomplish our objective the system must fulfill some properties. The steganogram must not look suspicious about something hidden in it. The capacity of the carrier of embedding must be enough to transport the secret message without being perceptible. The algorithm employed must be efficient and robust in order to not consume computational resources and it must take into account the intentions of detecting and destroying steganographic content.

Our approach is divided in six phases (see figure 1): embedding executable code in a carrier (BMP image), installing a steganographic parser at the system target, delivering a steganographic image to the target, detecting the steganogram at its arrival, extracting executable code from image and finally executing the information hidden (executable code).

In the embedding executable code process in a BMP image we implement a substitution algorithm that substitutes 1, 2, 3 or 4 LSB depending on the users' choice.

In order to extract and run the executable code, the steganographic parser must have been previously installed; this is the hard part of our proposal. This can be achieved in several ways. We can send to the system's users a trojan program and wait that some user executes it. Another solution is to design and use a remote exploit to install the parser.

Once the parser installed in the system, it must keep hidden and not raise any suspicions. The behavior of the system most bee as normal as if nothing new has occurred. By the other hand the system needs a way to let the steganographic parser analyze every picture visualized in the system. The system modifies some keys in the registry of system configuration, to accomplish the previous requirements.

Delivering the steganographic image is the easy part because it could be in a web page or in an e-mail, etc.; the images are the kind of files that circulates in computer systems.

Detecting the steganographic material is accomplished by the steganographic parser installed previously. It looks for a specific mark inside the file that distinguishes it from a normal image.

Extracting information from BMP file is folding out LSB used in the embedding process.

Executing hiding information could be in a different fashion way, creating a new thread, or process, it depends on the purpose of the hidden executable code.

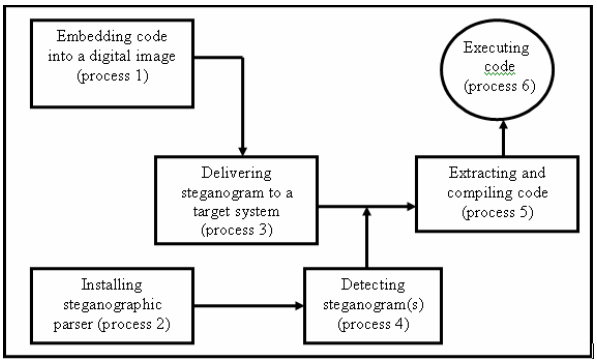

Fig. 1. Steganography executing code mode kernel

6 Experiments and results

We designed and implemented an application that is capable to hide executable a BMP file. The application is able to install the parser in the local host. We focus on BMP format because it has a plain structure and facilitates its implementation. Steganography experts recommend using shades of grey. Grey scale images are preferred because the shades change very gradually between its elements. This increases the images ability to hide information. However we decided color images in order to being more innocuous due to the fact that most users prefer color images.

We used substitution LSB steganographic technique in order that carrier size does not change that much, using data pixels. The information than can be hidden in an image file depends on how many bits per byte were used to incrust in the carrier.

All our experiments were done over Windowx XP Professional. The steganographic parser is installed in the path c:\windows\system with the name system32b.exe for being innocuous. The added key to install the parser was My_PC\HKEY_CLASSES_ROOT\Applications\system32b.exe\shell\open\command with value C:\WINDOWS\system\system32b.exe "%1". The modified key to register the parser was My_PC\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.bmp. With the sub key Application with value system32b.exe.

The embedded executable code used in every experiment was a typical Windows calculator application with a size of 115 200 bytes.

In figure 2 we show a screenshot of the application. The first option gives the possibility to incrust in 1, 2, 3 or 4 least significant bits on a BMP image. The choice depends on how much degradation is allowed. The next options allow selecting the original carrier, the file containing the executable code and the desired name for the steganogram. The interface counts with an option to activate/deactivate the steganographic parser. This last option allows us to test the system's behavior with and without an steganographic parser.

The application was developed in C language because its power in order to manipulate bits of the BMP image file. Firstly the application incrusts a mark (for example "GR") to distinguish a steganogram from a typical image in the first bytes of data pixels, and then incrust the size of the executable code. If the executable code is bigger than the data pixel image it expands the steganogram in order to fit the executable code. Only in this case the steganogram grow more in size than the carrier, in all the others cases the size remains the same. At first the application assures that the steganographic parser exists and has been properly installed in the host system if it is not, proceeds to install this one.

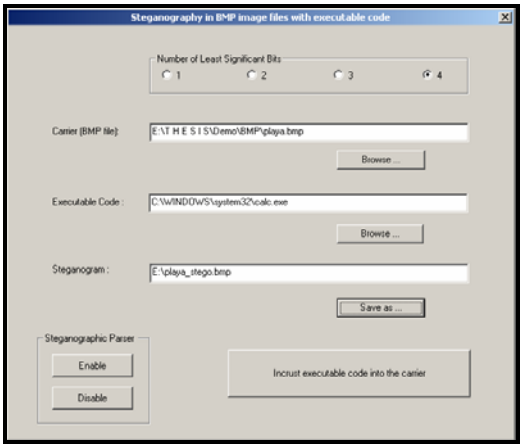

Fig. 2. Application developed interface

In order to test our system we select different images. The image carrier was selected for containing a variety of colors with a size of 342 294 bytes. Using 1 LSB the code needs 921 600 bytes (115 200 * 8) for being carried. The result steganogram can be seen in figure 3.

Fig. 3. Image with 1 LSB modified

Using 2 LSB the code needs 460 800 (115 200 * 8)/2 bytes for being carried. The result steganogram can be seen in figure 4.

Fig. 4. Image with 2 LSB modified

Using 3 LSB the code needs 307 200 (115 200 * 8)/3 bytes for being carried. The result steganogram can be seen in figure 5.

Fig. 5. Image with 3 LSB modified

Using 4 LSB the code needs 230 400 (115 200 * 8)/4 bytes for being carried. The result steganogram can be seen in figure 6.

Fig. 6. Image with 4 LSB modified

The results indicate that using 1 and 2 LSB results in a steganogram almost imperceptible to notice differences between original carrier and steganogram. However, using 3 and 4 LSB results a suspicion steganogram because it decreases the quality of the images.

Execution of executable code was successful in each case, as we can see in figure 7.

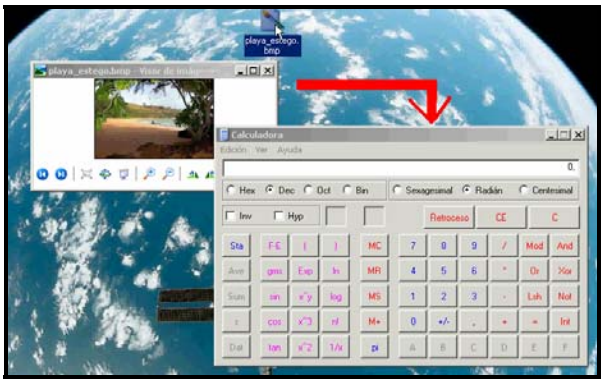

Fig. 7. Execution of calc.exe at the time visualizing 4 LSB steganographic image

The experiments were performed in other kind of images. For example images that present obscure colors like the one in the figure 8, present not degradation at all, even with its 4 LSB modified as in the case of figure 9. It is evident that with this kind of images is possible to conceal more information without raise suspicious unlike images with clear colors.

Fig. 8. BMP image with obscure colors

The results indicate that these images not alter its size and quality and hence they are the most appreciable candidate to be used in order to hide information.

Fig. 9. BMP image with image with obscure colors modified in 4 LSB

7 Conclusions and future work

In this paper, we have presented a mechanism to use steganography in order to propagate executable code. We have developed an application to demonstrate that this is possible. The application allows embedding and extracting executable code from pictures in BMP format.

The BMP format is ideal to incrust data because of its excessive redundancy. The embedding capacity results indicate that pictures with obscure colors are the ideal medium to transport executable code without making them look abnormal in anyway. They present more capacity than clear images.

We showed a practical example hiding the code of windows calculator in different images. We proved that this code can be extract when the user selects the image carrying the code.

The major breakthrough in our approach is that this schema could be used to different kind of applications not necessarily to spread malicious code but also every kind of code. For example could be used as a way to improve copyright, executable code might be scanning, looking for anomalies in the way we use commercial software or even used as a way for authentication from both sides client and server applications o media.

Steganography of executable code is a vast field waiting to explore. One of our future works involves apply our system to assure that the code was not altered in its journey to a target system. Another area of exploitation is how can be install steganographic parsers without the intervention of a user in the system affected.

It is necessary looking for ways of compression and cipher the executable code before its embedding in the image file.

BMP format is really accepted in the traffic in Internet but because of its size is needed to analyze other image format files that can be spread more easily than BMP like GIF and JPEG, achieving this could reach a major degree of propagation than only with users of the BMP format.

Parallel to our work of hiding executable code in image file formats, is necessary to design and implement schemes of detection. The schemes must be able to detect executable code in mages to avoid the dangerous that this technology could bring at the computer community.

References

1. Cole, E., Hiding in Plain Sight: “Steganography and the Art of Covert Communication”, Canada: 2003 Wiliey Publishing
2. N.F. Johnson, Z. Duric, and S. Jajodia, “Information hiding: Steganography and water-marking - attacks and countermeasures,” Kluwer Academic Publishers, 2000.
3. Enrique Cauich, Roberto Gómez, Ryouske Watanabe, “Data hiding in identificartion and offset IP fields”, LNCS Springer Verlag, Fifth IEEE International Symposium and School on Advance Distributed Systems, ISSADS 2005, January 24-28 Guadalajara, Jalisco, México
4. Chandramouli, R. Memon, N “Analysis of LSB based image steganography techniques Proceedings Image” Processing, vol 3, Greece, 2001. pp 1019-1022
5. C. Cachin, “An information-theoretic model for steganography,” Proc. 2nd Information Hiding Workshop, vol. 1525, pp. 306-318, 1998.
6. Sung Deok Cha, Gum Hen Park, Heung Kyu Lee. “Solution to the On-Line Image Down-grading Problem”, Annual Computer Security Applications Conference '95, Dec.13-15 1995. pp. 108-112.
7. A. Westfeld and A. Pfitzmann, "Attack on Steganographic Systems", Lectures Notes in Computer Science, vol. 1768, Springer-Verlag, Berlin, 2000, pp. 61-75
8. Jessica Fridrich lab: J. Fridrich, M. Goljan, R. Du, "Reliable Detection of LSB Steg-anography in Color and Grayscale Images", 2002
9. Katzenbeisser, S., Petitcolas, F., “Information Hiding Techniques for Steganography and Digital Watermarking”, 2000 Artech House, Inc.
10. Cochran, J., “Steganographic Computer Warfare”, Thesis, USAF, Air Force Institute of Technology, E.E.U.U., 2000.
11. Prakash; Nagesh; Singhal. A Possibility of Steganographic Trojan Installer. Network Security, IIT- Kanpur, Techkriti. 2002.
12. Rogers, M. Steganographic Trojans DEF-CON X.
13. Microsoft Security Bulletin MS04-028, Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution. <http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx> 2004.
14. Alcopaul, EBCVG #2 magazine 2002.