情報セキュリティ　第８回実習

Steganography

担当：鷲見和彦

今回の実習では、Steganography を C 言語で実装した事例を取り上げ、そのコード解析、コンパイル、実行、カバーメディアへのテキスト埋め込み、テキストの復元を実行する。今回は、Paul Macklin による Steganography のアルゴリズムを試す。

１．ソースコードの展開

i. BMP ファイルのライブラリを http://prdownloads.sourceforge.net/easybmp/EasyBMP_1.06.zip?download から取得する

ii. Steganography のプログラムを http://easybmp.sourceforge.net/downloads/samples/Steganography.zip から取得する

iii. それぞれを、作業用ディレクトリ（例えば、08-exercise¥Paul_Macklin¥ というディレクトリを作っておく）で、展開すると EasyBMP_1.06 と Steganography というサブディレクトリに展開される。

２．ソースコードの解析

i. Steganography¥Steganography.cpp をエディタ（Meadow や秀丸）で閲覧する。

ii. Main() 関数を探し、プログラムのアルゴリズム説明（*1）を見ながら、コマンドライン引数の処理を探し、encode / decode 処理の分岐を見つけ、それぞれ encode 時と decode 時の処理を探す。

(*1) http://easybmp.sourceforge.net/steganography.html

iii. Encode 時の処理について（*1）の記述と比較して、アルゴリズムの実装を確認する。隠蔽するテキストファイルの文字 1byte が 8bit に分解され、隣接する 2 画素の RGBA RGBA 値に 1bit づつ埋め込まれていることを確認せよ。

iv. 同様に、Decode 時の処理についてアルゴリズムの実装を確認する。

３．ソースコードのコンパイル

i. Cygwin コマンドプロンプト（Windows cmd.exe でも可）で、EasyBMP_1.06 ディレクトリへ cd し、g++ -g -c EasyBMP.cpp を実行して、オブジェクト EasyBMP.o を作成する。

-g オプションは、動作不良の際にデバッグできるようにシンボルテーブルを残すという意味。

```
cd EasyBMP_1.06
g++ -g -c EasyBMP.cpp
```

これにより EasyBMP.o というオブジェクトファイルが出来る。

ii. プログラム本体のディレクトリへ移動

```
cd ../Steganography
```

iii. 必要なヘッダファイルと、EasyBMP.o をコピー

```
cp –p ../EasyBMP_1.06/*.h .
cp –p ../EasyBMP_1.06/*.o .
```

iv. コンパイル

```
g++ -g –o Steganography.exe Steganography.cpp EasyBMP.o
```

v. カバーメディア covermedia.bmp, 隠すテキストファイル sample.txt を用意する。

```
cp ../covermedia.bmp ../sample.txt .
```

vi. エンコード

```
./Steganography.exe -e sample.txt covermedia.bmp stegomedia.bmp
```

vii. 埋め込み情報をバックアップしておく。(デコード時に上書きされるので)

```
mv sample.txt sample.txt.orig
```

viii. デコード

```
./Steganography.exe -d stegomedia.bmp
```

ix. sample.txt が復元され、その内容が同じであることを diff で確認。(diff は、二つのファイルに差が無ければ、何も出力しない。差があれば、違いを出力するコマンドです。)

```
diff sample.txt sample.txt.orig
```

4．ステゴメディアの確認

ステゴメディア stegomedia.bmp とカバーメディア covermedia.bmp との差を GIMP などのツールを用いて確認せよ。

2 枚の画像の差を調べる方法のヒント：

GIMP を使い、レイヤー間の絶対値差を表示する
https://isp-image-d.hatenadiary.org/entry/20110328/1301291785

ImageMagic の composite を使う
https://blog.mirakui.com/entry/20110326/1301111196

以上

参考情報：アルゴリズム説明(*1)

EasyBMP Code Sample: Steganography

Note:

If you're looking for the old version of this page (with some more primitive steganographic techniques), click here.

Summary:

We present a basic example of steganography: the hiding of information within an image. After giving some information on how to do it, we give source code that was developed with the EasyBMP library.

Intro:

Steganography is the process of hiding information within an image file in such a way that it is nearly indetectable. It should at least be indetectable to the human eye. Ideally, it should even be hard for a computer to detect the information without getting a lot of "false positives" when scanning many images.

Information is hidden in one of these two images. Can you determine which one?

Figure 1: One of these images has hidden information. Can you determine which one?

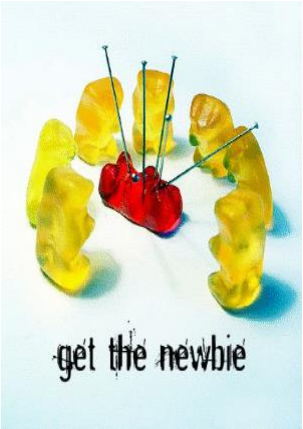

Figure 2: This is the image that was hidden in the previous figure. Picture-in-picture, if you will. :-)

In Figure 2, you can see the image that was hidden in Figure 1. Furthermore, click on the picture to download both the original and altered images. As we can see, hiding data in a photograph is particularly effective, since the hidden data is often smaller than the noise and spatial patterns of the photo.

Steganography is not encryption. The point isn't to make the information impossible to decrypt, but rather to hide the data in the middle of innocuous data. Usually, this is done by modifying the image pixels in some way, e.g., by slightly modifying the color of the pixels. In a certain sense, this is "security by obscurity," although we're merely relying upon the hidden nature of the data, rather than a broken cytrographic method.

However, the data can be encrypted before hiding it in the image. In such a scenario, not only would the data be hard to be detect, but even if the steganographic method were determined, its contents would then need to be cracked. In this sense, cryptography and steganography are complementary: steganography hides data but doesn't secure it, and cryptography secures data but doesn't hide it. Combining them can be quite potent.

How it Works

The core of any steganographic method is how it encodes a single byte (8 bits) of information within the image. Some methods take advantage of the file structure of the image and hide it in special data fields. For example, in the BMP file format, the offset between the file information and pixel data can be manually specified. This presents interesting possibility of hiding an entire file between the file information and the pixel data without altering the image at all. In practice,

such methods are impossible to detect visually but easy to detect with a computer: if the file size is larger than the minimum size necessary for the image size and color depth (according to the file format specification), then it probably included hidden data.

Other steganographic methods hide data by slightly modifying the pixels of the actual image by small amounts. Typically, the modification is done by changing the least significant bit (or bits) of the red, green, blue, and applicable, alpha channels of one or more pixels. This is how we proceed in our sample program.

A Sample Steganographic Method

Any byte is a number N from 0 to 255. This means that we can expand it in binary: if $r_1, g_1, b_1, a_1, r_2, g_2, b_2, a_2 \in \{0,1\}$, then we can write N as

$$N = r_1 + g_1\, 2 + b_1\, 2^2 + a_1\, 2^3 + r_2\, 2^4 + g_2\, 2^5 + b_2\, 2^6 + a_2\, 2^7$$

Then if (R_1,G_1,B_1,A_1) and (R_2,G_2,B_2,A_2) are two adjacent pixels in a 32 bpp image, we overwrite them with the new pixels in Figure 3.

% = modulus operator

$(R_1,G_1,B_1,A_1) - (R_1,G_1,B_1,A_1) \% 2 + (r_1,g_1,b_1,a_1)$

$(R_2,G_2,B_2,A_2) - (R_2,G_2,B_2,A_2) \% 2 + (r_2,g_2,b_2,a_2)$

Figure 3: The algorithm used in our steganographic program.

Suppose, for example, that two adjacent image pixels are (255,255,255,0) and (255,255,255,0), and we wish to hide the data $N = 18$. Then by our algorithm, we hide the data as in Figure 4. Notice how little the pixels change to hide the data; to the human eye, both pixels are still white.

$N = 18$
$r_1 = N \% 2 = 18 \% 2 = 0$
$T = r_1 = 0$
$g_1 = (N - T)/2 \% 2 = (18 - 0)/2 \% 2 = 9 \% 2 = 1$
$T = T + 2 * g_1 = 0 + 2*1 = 2$
$b_1 = (N - T)/4 \% 2 = (18 - 2)/4 \% 2 = 4 \% 2 = 0$
$T = T + 4 * b_1 = 2 + 4*0 = 2$

$a_1 = (N - T)/8 \ \% \ 2 = (18 - 2)/8 \ \% \ 2 = 2 \ \% \ 2 = 0$
$T = T + 8 * a_1 = 2 + 8*0 = 2$

$r_2 = (N - T)/16 \ \% \ 2 = (18 - 2)/8 \ \% \ 2 = 1 \ \% \ 2 = 1$
$T = T + 16 * r_2 = 2 + 16*1 = 18$
$g_2 = (N - T)/32 \ \% \ 2 = (18 - 18)/2 \ \% \ 2 = 0 \ \% \ 2 = 0$
$T = T + 32 * g_2 = 18 + 2*0 = 18$
$b_2 = (N - T)/64 \ \% \ 2 = (18 - 18)/64 \ \% \ 2 = 0 \ \% \ 2 = 0$
$T = T + 64 * b_2 = 18 + 4*0 = 18$
$a_2 = (N - T)/128 \ \% \ 2 = (18-18)/128 \ \% \ 2 = 0 \ \% \ 2 = 0$

So, the new pixels are

(255,255,255,0) – (255,255,255,0) % 2 + (0,1,0,0)
= (255,255,255,0) – (1,1,1,0) + (0,1,0,0)
= (254,255,254,0)

(255,255,255,0) – (255,255,255,0) % 2 + (1,0,0,0)
= (255,255,255,0) – (1,1,1,0) + (1,0,0,0)
= (255,254,254,0)

Figure 4: An example of hiding the data $N = 18$ in two adjacent white pixels.

In addition to the actual file data, we encode the following information prior to the file data using the same steganographic technique:

EasyBMPstego (12 bytes)
[Number of characters in output filename] (2 bytes, given by $FNS = a + b*255$)
Output filename (VNS bytes)
[Number of hidden bytes] (4 bytes, given by $FS = a + b*255 + c*255^2 + d*255^3$)

Extracting data from an image is done similarly. First, we check to see that EasyBMPstego has been encoded in the first 24 pixels. If so, we then proceed to read the next four pixels to extract the size of the filename, read the actual file name, the size of the file data in the following 8 pixels, and the file data itself.

Potential Improvements

This method could be improved in several ways to increase data storage capacity and better prevent detection.

Improving Storage Capacity

Currently, this method hides one bit of data per red, green, blue, and alpha channel in two adjacent pixels. This means that 1 byte of data is hidden per 8 bytes of image data. (0.5 bytes per pixel) If the alpha channel is completely unimportant, one could hide the entirety of the data in the alpha channel. This would improve the storage capacity to 1 byte of hidden data per 4 bytes of image data. (1 byte per pixel) Furthermore, there would be no visible distortion of the image, assuming that the alpha channel isn't important. (But this change would make the method easy to detect without further changes.)

Another way to improve the storage capacity would be to modify 2 bits per red, green, blue, and alpha channel of a single pixel. Or a variant would be to modify 1 bit per red, green, and blue channel and 5 bits of the alpha channel. This would also store a single byte of data in a single pixel, with little distortion of the visible pixels and moderate distortion of the alpha channel.

An interesting alternative approach would be to modify one bit of the red, green, and blue channels and all 8 bits of the alpha channel. This would allow the storage of 11 bits per pixel, which is more than one byte. (1.375 bytes per pixel) Thus, 11 bytes of data could be stored 8 adjacent pixels. This is probably the best uncompressed storage that can be attained in an image where the alpha channel isn't important without being visually detectable.

Finally, a lossless compression (e.g., zip or gzip) could be applied to the data prior to encoding it in the image.

Preventing Detection

Our method can most easily be detected if the hidden data requires fewer pixels than given by the image. For example, suppose we are hiding data in a white image, and suppose we only require half the pixels to hide the data. Then the bottom of the image will be pure white, and the top won't be. While the human eye won't detect this difference, a computer program could very easily. A human could detect the data very easily by using a paint fill tool on the bottom of the image: only the bottom would fill to a solid color.

The simplest solution to this is to never use images with large patches of a single color; photographs are best suited to steganography. Another way to ameliorate this problem is to hide random data after the true data, so that all pixels are modified.

This is particularly important when hiding data in the alpha channel, when the alpha channel is not normally important. In such a case, nonzero alpha channel data would reveal the presence of hidden data. Again, the best protection is to hide random data in the alpha channel of the unused pixels. If all 8 bits of data are hidden in the alpha channel, then the random data should also by 8 bits.

Source Code and Demo Programs:

I used the EasyBMP library to write these algorithms. It only took a few minutes to write the code that changes the pixels to encode the information; the majority of the time was spent determining how best to provide file information on the hidden data and on the steganographic algorithm itself. This is exactly the point of EasyBMP: to free yourself from worrying about image format details so you can focus on your real work. :-)

This download includes the GPL'ed source code and a win32 executable. You'll still need to download EasyBMP to compile it. This program was designed to be cross-platform and cross-architecture-compatible with a variety of compilers.

File	Description	License	Updated
Steganography.zip	EasyBMP Steganography Program	GPL 2.0	2-3-2006

Usage:

Type Stego -h or simply Stego to get full help information and usage.

To hide aFileToHide.abc in the image BaselineImage.bmp, do this:

Stego -e aFileToHide.abc BaselineImage.bmp OutputImage.bmp

The result will be OutputImage.bmp, a 32 bpp BMP file.

To extract any hidden data in the image file SomeImage.bmp, do this:

Stego –d SomeFile.bmp

Stego will process the file, and if hidden data is detected, it will inform you and output to the original filename.

Some Last Information

This program is provided for fun and personal use only and to demonstrate the basic ideas of steganography. It certainly isn't secure, and you shouldn't rely upon it for security. Furthermore, it isn't intended for use in conducting illegal activity. I have no doubt that folks at the NSA and elsewhere could detect and extract the information in these images very easily, and so I wouldn't even think about doing something illicit with them!!