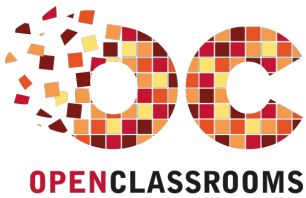
Dissimuler un texte dans une image

Par Cid



www.openclassrooms.com

Dernière mise à jour le 17/02/2011

Sommaire

Dissimuler un texte dans une image

Par  Cid

Mise à jour : 17/02/2011
Difficulté : Facile

Salut tout le monde !

Prêts pour une incursion dans le monde de la stéganographie, dans l'art de la dissimulation ? Alors allons-y !

Dans ce tutoriel, vous allez apprendre à dissimuler un message quelconque au sein d'une image Windows bitmap BMP (eh oui, c'est malheureusement une limitation bien embêtante pour ce genre de choses, mais nous verrons pourquoi on ne peut pas utiliser d'image JPEG... mais par contre, on peut utiliser (sous certaines conditions !) des images PNG !). Prêts à tenter l'aventure ? !

Le code que je vais donner est du code PHP, parce que c'est le langage que je maîtrise le mieux. Il est par contre très facile de porter le code en C++ ou en Java (je l'ai par ailleurs réécrit en C++ sans jamais en avoir fait auparavant, donc ce n'est vraiment pas dur). Le seul problème qui se pose alors est que la console Windows gère **très** mal les accents ; c'est pourquoi une page HTML semble plus appropriée comme moyen d'affichage.

Sommaire du tutoriel :

L'idée de départ

L'idée est la suivante. On a un texte en clair (donc pas chiffré du tout, et lisible par n'importe qui) que l'on veut dissimuler « dans » une image (un fichier bitmap). Évidemment, il ne s'agit pas d'écrire le texte en plein milieu du dessin, ni d'ajouter du texte en commentaire dans l'image (comme les champs EXIF des images JPEG). Non, ça ne serait pas super discret. L'idée est plutôt de modifier les bits de poids faible des pixels par un bout de lettre…

Bit de poids faible ? Quésako ?

Ah oui, un petit rappel s'impose je crois.

La représentation binaire

Depuis… allez… le CP, vous savez certainement compter **en base 10**. Prenons le nombre « 1992 ». Vous savez tous que la valeur de ce nombre s'obtient comme suit :

$1992 = 2*10^0 + 9*10^1 + 9*10^2 + 1*10^3$

C'est-à-dire :

1992 = 2+9*10+9*100+1*1000 (Oui j'ai fait exprès de commencer par la droite, vous verrez rapidement pourquoi.)

Remarquons la chose suivante : si je change la valeur du chiffre tout à droite du nombre (ici le 2), la valeur du nombre ne changera pas de beaucoup (si je change le 2 par un 9, j'ajoute 7 à la valeur du nombre), alors que si je change la valeur du chiffre tout à gauche (ici le « 1 »), la valeur du nombre changera beaucoup (jusqu'à 8000 de différence tout de même).
Bien.

Maintenant, place à la représentation binaire. En fait, en binaire, la valeur d'un nombre s'obtient de façon tout à fait analogue. Par exemple, le nombre dont l'écriture binaire est 100101000101 se décompose comme suit (je commence par la droite encore une fois, dans ce cas-ci c'est plus simple pour ne pas s'emmêler avec les puissances) :
$100101000101 = 1*2^0 + 0*2^1 + 1*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 1*2^6 + 0*2^7 + 1*2^8 + 0*2^9 + 0*2^{10} + 1*2^{11}$

Ouf ! D'ailleurs, la même remarque que tout à l'heure s'impose : si je change la valeur de un ou deux voire trois chiffre(s) tout à droite, la valeur du nombre ne changera pas de beaucoup (au pire de 4 en changeant 2 chiffres, 8 en en changeant 3). Ce sont ces chiffres-là, dans une telle représentation binaire, qui sont appelés « bits de poids faible », tout simplement parce que si l'on change leur valeur, la valeur du nombre ne sera pas changée de beaucoup. A contrario, les bits les plus à gauche sont appelés « bits de poids fort », pour une raison analogue…

Bien, maintenant, l'idée est la suivante : on associe à chaque lettre une valeur, d'après la table ASCII, et ensuite…

Hé ! Stop ! C'est quoi la table ASCII ?

Argh, je m'attendais à cette question. En fait, c'est très simple. L'idée est d'associer à chaque caractère un nombre. Évidemment, on ne pouvait pas tout simplement prendre la position des lettres dans l'alphabet (A=1, B=2, …). Comment différencier majuscules et minuscules ? Et comment représenter les signes de ponctuation ? C'est pour cela que l'on a créé la table ASCII, qui justement, associe à chaque nombre de 0 à 127 un « caractère ». Notez que tous les caractères ne sont pas obligatoirement « imprimables » ; en sus des chiffres et des lettres, il existe aussi des caractères de contrôle, des trucs bizarres comme des carillons, etc.

Bien bien, mais pourquoi aller jusqu'à 127 seulement alors qu'on peut stocker 256 valeurs différentes dans un octet ?

Très bonne question, encore une fois. En fait, le code ASCII a été introduit alors que les ordinateurs n'en étaient qu'à leur début. En particulier, les transmissions n'étaient pas vraiment très fiables à cette époque. Quel est donc le rapport avec la choucroute ? En fait, les nombres de 0 à 127 ont tous un point commun... leur bit de poids fort (donc celui tout à gauche normalement) vaut 0. Ce bit était appelé bit de contrôle, et servait à vérifier que la transmission avait eu lieu sans trop de cafouillage…

Autre petit point intéressant. L'ASCII est ce qu'on appelle un encodage de caractères… et il en existe d'autres. En particulier l'Unicode (ou UTF-8), qui gère aussi les alphabets non latins (russe, arabe, grec, …) et bien plus de caractères accentués. Mais pour se faciliter les choses au début, on va se limiter à du ASCII pour notre texte, tout simplement parce qu'en ASCII, un caractère = un octet. Pour ceux que ça intéresse, voici la table ASCII qui donne la correspondance entre un nombre et un caractère : table ASCII . Une version française peut être trouvée ici.

Tous les problèmes ont été réglés ? Bien ? Dans ce cas, continuons notre explication…
On a donc vu que l'on peut associer à chaque caractère de notre message un chiffre (en théorie inférieur à 127, mais on va considérer que l'on utilise l'ASCII « étendu » qui va jusqu'à 255 — simple détail). Ce chiffre va pouvoir être représenté sous forme binaire et il tient dans un octet.
Prenons un exemple concret. En ASCII, l'apostrophe « ' » est codée par le chiffre 37. Ainsi, on a l'association « ' » = 37. De plus, en binaire, 37 s'écrit 00100101, d'où l'association « ' » = 00100101. Bien compris ? Passons donc à l'étape suivante.

On va maintenant « couper » ce chiffre en petits bouts. On va dire que la longueur de ces bouts va valoir 2 (on peut aussi utiliser 4, mais on risque de trop détériorer l'image. Une longueur de 3 complique tout (même si c'est faisable) et une longueur de 1 demande trop de place). Ainsi, en découpant 37, ça nous donne ça :

00100101 = 00 10 01 01

Bon, maintenant on va cacher ces bouts dans notre image. Mais comment faire ?

Le stockage de l'information dans une image bitmap

Comme vous le savez tous, un pixel est défini par 3 (ou 4 avec la transparence, le fameux canal alpha) composantes : une composante rouge, une bleue, et une verte ; c'est le fameux RVB (il existe d'autres représentations, notamment la représentation CMJN pour l'imprimerie, mais elle ne nous intéresse pas ici). Une image de type BMP (Windows bitmap) est, quant à elle, composée comme suit : après un *header*, qui indique notamment les dimensions de l'image, suit une suite d'octets, qui, pour chaque pixel, en commençant en bas à gauche, indique successivement les valeurs de la composante bleue, puis verte, puis rouge (oui, c'est le sens inverse de RVB). Après 3 octets commence le codage du deuxième pixel, et ainsi de suite. De plus, d'après Wikipédia (version anglaise, voir l'exemple donné), nous savons que, dans une image BMP standard à 24 bits (c'est-à-dire un octet pour le bleu, un octet pour le vert, un octet pour le rouge), le *header* a une taille fixe de **54** octets. Nous avons toutes les informations pour commencer notre insertion dans l'image…

Euh comment ça, on a toutes les informations ? À quoi ça nous sert tout ça ?

Bon, prenons un exemple. On va prendre n'importe quelle image bitmap, l'ouvrir avec un éditeur hexadécimal (comme Ghex sous Linux), sauter le *header* et regarder ce qu'il nous donne :

Citation : ghex

00010110 00010100 00011110 00010101 …

(Oui normalement vous aurez une représentation hexadécimale et non pas binaire, mais en théorie le programme donne *aussi* une représentation binaire.)
Chaque groupe de 8 bits (donc chaque octet) indique (en binaire) les quantités respectives de bleu, vert et rouge du premier pixel (en bas à gauche pour rappel). Ainsi, le premier pixel aura une intensité de bleu de 22, une intensité de vert de 20 et une intensité de rouge de 30. Si on définit une telle couleur dans Paint, on aura très probablement une couleur très foncée...Le quatrième octet, quant à lui, indique la quantité de bleu du deuxième pixel. Mais bon, à vrai dire, on s'en fiche un peu de tout ça. Nous ce qu'on veut, c'est dissimuler notre texte dans cette image. Comment allons nous donc faire ? Substituer tout simplement les bits de poids faible de l'image (c'est-à-dire les bits les plus à droite pour chaque couleur) par les morceaux de notre caractère ! En pratique, ça donne ça :

' = 37 = 00100101 = 00 10 01 01

Une fois qu'on a découpé l'octet représentant notre caractère, on va modifier les bits de poids faible de l'image en conséquence. Ce qui nous donne donc trois nouveaux octets :

Citation : ghex

00010100 00010110 00011101 00101001 …

Vous voyez l'idée ? On échange les deux bits de poids faible de chaque couleur par deux bits correspondant à une partie du caractère. Ainsi, on ne change que très peu l'image finale (l'œil humain est incapable de voir une différence de 4 sur une teinte de 256 tons), mais on peut très facilement retrouver le message final : il suffit de lire tous les deux bits de poids faible de chaque octet, et de les assembler en octets pour reconstituer les caractères du message.

Pour ceux qui se demandent « et en prenant une image JPEG ? », la réponse est la suivante. Contrairement au « Windows bitmap », le format JPEG compresse l'image. Cela signifie qu'on n'a plus l'équivalence « un octet = une valeur de couleur d'un pixel ». Donc, si on change les bits de poids faible des octets d'une image JPEG, on n'a *aucune* idée de ce qu'on modifie. On pourrait alors penser qu'il suffirait de décompresser l'image JPEG, de changer les bits de poids faible pour chaque pixel, puis de compresser à nouveau. Sauf que voilà, la compression modifie la valeur de chaque teinte de chaque pixel. La variation est de 2 en moyenne et peut atteindre 4 voire plus. Difficile (parfois) à distinguer pour un œil humain, mais notre message devient alors illisible…

Quelques considérations mathématiques

Avant de nous lancer dans le code, il reste un ou deux problème(s) à régler.

Premier problème : comment modifier les bits de poids faible ?

Très bonne question. En d'autres termes, comment arriver à partir de 00011110 et de 01 à 00011101 ?

…

…

Vous ne voyez pas ? Avec les modulos pardi ! Ou plus exactement, les restes de la division euclidienne…

Petite explication. Lorsque vous avez le nombre dont l'écriture décimale est 1992 et que vous voulez avoir 1990, vous faites comment ? Vous enlevez 2, tout simplement. Pourquoi 2 ? Parce que c'est le reste de la division euclidienne de 1992 par 10 (en effet, 1992 = 10*199 + 2). Bien. Si maintenant on veut avoir 1900 à partir de 1992, on va tout simplement enlever 92. Eh oui, vous l'aurez deviné, 92 est le reste de la division euclidienne de 1992 par 100. Eh bien, en binaire, c'est la même chose ! Pour passer de 00011110 à 00011100, il suffit de soustraire à 00011110 le reste de la division euclidienne de 00011110 par 4 (soit 10 en binaire, c'est-à-dire 2), et pour passer de 00011100 à 00011101, il suffit d'ajouter 01 (donc 1, en gros…). Ce n'est pas plus dur que ça. Ah oui, dernier détail, l'opérateur modulo (qui, en réalité, donne le reste de la division euclidienne d'un chiffre par un autre) est le signe « % » dans la plupart des langages (en particulier dans ceux basés sur le C : C++ mais aussi PHP, Java, etc.).

Petite remarque : pour tous ceux qui sont à l'aise avec les opérateurs de bits, il est également possible de résoudre ce petit problème avec (petit rappel ici : Introduction aux opérateurs de bits). Mais malheureusement, il faut aussi le faire en deux étapes. En effet, 00011110 & 11111100 ($=2^8 - 2^2$) donnera 00011100, et il suffit ensuite de rajouter les deux bits intéressants (ou de faire une opération de OU binaire — ça revient au même). Mais bon, c'est juste pour la petite histoire, hein… ça ne risque que de compliquer le code à la fin…

Comment lire les bits de poids faible d'un octet ?

Avec la méthode vue précédemment, vous devriez avoir deviné… Il suffit de prendre le reste de la DE de la valeur de l'octet par 4, et le tour est joué !

Une fois que l'on a récupéré toutes les valeurs des bits de poids faible, comment fait-on pour retrouver la valeur de l'octet du caractère codé ?

Une manière simple est de tout stocker dans une chaîne de caractères, puis de passer par des fonctions qui donnent la valeur décimale d'un nombre stocké en représentation binaire. En PHP, il s'agit de la fonction *bindec()*. À noter que la fonction *decbin()* fait exactement l'inverse (on passe d'un nombre décimal à un nombre en base 2). On aura besoin des deux fonctions à un moment ou un autre de notre programme. Et si vous voulez écrire le programme en un autre langage, il faudra coder vous-mêmes ces deux fonctions (avec des *substr()* ce n'est pas très dur…), voici ce que ça pourrait donner en C++ :

Secret (cliquez pour afficher)

Code : C++

```
int bindec(string binaire)
{
        int rep_decimale=0;
        string bit, zeros = "";

        if(binaire.size() < 8) // Si on se retrouve avec moins de
8 caractères.
        {
            zeros.append(8-binaire.size(), '0');
            binaire = zeros.append(binaire);

        }

        for(int i=0;i<=7;i++)
        {
            bit = binaire.substr(i,1);
            if(bit == "1")
            {
                rep_decimale += pow(2, 7-i);
            }
        }
```

```
        return rep_decimale;
}

string decbin(int decimal)
{
    string rep_binaire = "";

    if(decimal > 255)
    {
        cout << "Trop grand";
        return "0";
    }
    else
    {

     for(int i = 7;i>=0;i--)
 {
     if(decimal >= pow(2, i))
     {
         rep_binaire.append("1");
         decimal -= pow(2, i);
      }
      else
      {
          rep_binaire.append( "0" );
      }
         }

       return rep_binaire;
     }

}
```

Ces deux fonctions ne fonctionnent qu'avec les nombres dont la représentation décimale ne dépasse pas 255, donc les nombres tenant dans un octet… On ne veut pas plus en même temps (évidemment, on pourrait les coder de nouveau pour que les fonctions acceptent de plus grands nombres aussi, mais ce serait une perte de temps ici)…

À présent que tout est réglé… à nos éditeurs de texte préférés. Que le codage commence !

La partie Insertion

Commençons par la partie insertion du message dans l'image. Ce n'est pas forcément la plus simple, mais au moins, on pourra plus facilement tester si tout s'est bien passé. Parce qu'en fait, l'implémentation de l'algorithme pose quelques problèmes… Regardons le code suivant (du PHP parce que c'est plus simple à tester et qu'il n'y a pas de problèmes d'accents...) :

Code : PHP

```
<?php
$message = 'Top secret ultra confidentiel !';
$octet_decoupe = array();

for($i=0;$i<strlen($message);$i++)
{
     $caractere = $message[$i];
     $valeur_octet = ord($caractere);
 $octet_binaire = decbin($valeur_octet);
 $octet_decoupe = str_split($octet_binaire, 2);

 foreach($octet decoupe AS $partie octet)
```

```
 {
  //...
 }
}
?>
```

Rien de bien compliqué là-dedans. Il faut juste savoir que la fonction *ord()* donne le code ASCII d'un caractère, que *strlen()* indique la longueur d'une chaîne de caractères (oui, j'utilise une syntaxe un peu spéciale mais tout à fait valide en PHP comme en C ou en C++), et que *str_split()* découpe une chaîne de caractères en petits morceaux. Mais voilà, nous avons un petit problème. Essayons de prévoir ce que donnera ce code (du moins pour la première itération). Un T majuscule est codé par le nombre 84 en ASCII, ce qui correspond à 1010100 en base 2. Une fois que l'on aura découpé ça en morceaux, on aura le découpage suivant :

1010100 = 10 10 10 0

Aïe ! Vous voyez où ça coince ? Le dernier zéro se trouve tout seul. En effet, votre langage est intelligent, il ne va pas rajouter des zéros inutiles avant le premier chiffre. Mais nous, ça nous embête ! On le veut ce zéro, sinon on ne peut pas découper notre nombre correctement ! Heureusement que PHP nous fournit une fonction bien sympathique : str_pad ! (merci Cortexd :p)

Nous allons l'utiliser de la manière suivante :

Code : PHP

```
<?php
$octet_binaire = str_pad($octet_binaire, 8, '0', STR_PAD_LEFT);
?>
```

Je m'explique : le premier argument est la chaine qu'on veut modifier, le deuxième la taille désirée (on va utiliser 8 ou 2), le '0' indique qu'on rajoute des '0', et le dernier argument spécifie qu'on rajoute les zéros à gauche et non à droite. Evidemment, c'est un peu long à écrire à chaque fois, on utiliserai une macro en C ou en C++, mais en PHP on peut pas... donc on va le réécrire à chaque fois. Tant pis !

Modifions donc le code en conséquence :

Code : PHP

```
<?php

$message = 'Top secret ultra confidentiel !';
$lien = 'images/trucmuche.bmp';

$octet_decoupe = array();

// Là, ça commence à devenir intéressant.
$f_image = fopen($lien, 'r+b'); // On ouvre le fichier image, tout
simplement.
fseek($f_image, 54); // On se place après le
<italique>header</italique>.

for($i=0;$i<strlen($message);$i++)
{
     $caractere = $message[$i];
     $valeur_octet = ord($caractere);
 $octet_binaire = decbin($valeur_octet);

 $octet_binaire = str_pad($octet_binaire, 8, '0', STR_PAD_LEFT); //
La ligne nouvelle

 $octet_decoupe = str_split($octet_binaire, 2);

 foreach($octet_decoupe AS $partie_octet)
 {
```

```
    ...
  }
}

?>
```

N'avançons pas trop vite ! Alors, qu'est-ce qui a été rajouté ? Un appel à notre nouvelle fonction, soit, mais on commence aussi à manipuler un fichier. Oh-oh, là ça devient intéressant.
On doit commencer par sauter le *header*, c'est pourquoi on place notre curseur à la position **54**, et puis ensuite…

Pourquoi 54 alors que tu nous avais dit tout à l'heure que le *header* avait une taille fixe de 54 pixels dans une image BMP 24 bits ? Ce ne serait pas plutôt 53, vu que le premier octet compte comme 0 ?

Remarque pertinente, je dois avouer. En fait, pour cela, il faut comprendre le fonctionnement des curseurs de position en PHP (ceci est aussi valable en C ou C++). Si je le place à la position 0 (*fseek($f_image, 0)*), il sera placé **avant** le premier octet, de sorte que ce sera le premier octet qui sera lu. Donc, si je le place à la position 54, il sera placé **avant** le 55^{e} octet. Or, c'est bien le 55^{e} octet qui est le premier octet de l'image et non pas le 54^{e}…

Il faut absolument utiliser un mode d'ouverture r+ et non a+. Si on ouvre le fichier en a+, alors on ne pourrai pas modifier des octets dans le fichier mais seulement en rajouter à la fin du fichier, ce qui n'est pas ce qu'on veut...

Continuons donc cette partie dans le *foreach* (le reste est quasiment terminé en fait).

Code : PHP

```
<?php


 foreach($octet_decoupe AS $partie_octet)
 {


  $octet_image = fread($f_image, 1); // On récupère un seul octet,
sous forme de caractère.
  $octet_image = ord($octet_image); // On le convertit en nombre
grâce à la table ASCII.

  $octet_image -= $octet_image%4; // On rend les deux bits de poids
faible égaux à zéro. La ligne suivante est équivalente mais utilise
les opérateurs de bit.
  //$octet_image = $octet_image & 252;

  $partie_octet = bindec($partie_octet); // On reconvertit en base
10 pour pouvoir faire une addition.

  $octet_image += $partie_octet; // La deuxième étape

  fseek($f_image, -1, SEEK_CUR); // TRÈS IMPORTANT

  fputs($f_image, chr($octet_image)); // On écrit tout simplement
dans le fichier, en écrasant l'octet suivant. Le chr convertit le
nombre en caractère puisqu'en PHP, on insère des chaines de
caractères. En C, on aurait pu utiliser fputc(f_image, octet_image)
sans passer par une conversion

 }

?>
```

Voilà, l'essentiel du code est là-dedans. Tout est plutôt simple à comprendre, à part peut-être ce deuxième *fseek()*. Qu'est-ce qu'il fait là, lui ?
En fait, c'est tout simple. Le problème est le suivant : si on lit un caractère dans un fichier, le curseur avance d'une position. Si on écrit ensuite sans reculer le curseur (ce que fait l'appel à la fonction *fseek($f_image, -1, SEEK_CUR)*, l'argument « SEEK_CUR » signifiant que la position est donnée par rapport à la position actuelle), on écrase l'octet suivant, auquel on ne veut pas encore toucher… et ça donne un beau méli-mélo (un octet inchangé, un octet représentant l'octet précédant tel qu'il aurait dû être après modification, un octet inchangé, etc.). Bref, on fout en l'air toute l'image, bravo la discrétion ! C'est le seul piège. Donc voilà, en cadeau, le code final…

Code : PHP

```php
<?php

$message = 'Top secret ultra confidentiel !';
$lien = 'images/trucmuche.bmp';

$octet_decoupe = array();

$message .= chr(26);

// Là, ça commence à devenir intéressant.
$f_image = fopen($lien, 'r+b'); // On ouvre le fichier image, tout simplement.
fseek($f_image, 54); // On se place après le <italique>header</italique>.

for($i=0;$i<strlen($message);$i++)
{
     $caractere = $message[$i];
     $valeur_octet = ord($caractere);
 $octet_binaire = decbin($valeur_octet);

 $octet_binaire = str_pad($octet_binaire, 8, '0', STR_PAD_LEFT);

 $octet_decoupe = str_split($octet_binaire, 2);

 foreach($octet_decoupe AS $partie_octet)
 {

  $octet_image = fread($f_image, 1); // On récupère un seul octet, sous forme de caractère.
  $octet_image = ord($octet_image); // On le convertit en nombre grâce à la table ASCII.

  $octet_image -= $octet_image%4; // On rend les deux bits de poids faible égaux à zéro. La ligne suivante est équivalente mais utilise les opérateurs de bit.
  //$octet_image = $octet_image & 252;

  $partie_octet = bindec($partie_octet); // On reconvertit en base 10 pour pouvoir faire une addition.


  $octet_image += $partie_octet; // La deuxième étape

  fseek($f_image, -1, SEEK_CUR); // TRÈS IMPORTANT

  fputs($f_image, chr($octet_image)); // On écrit tout simplement dans le fichier, en écrasant l'octet suivant.

 }
}

fclose($f_image);

?>
```

Dans le code final, deux lignes supplémentaires sont apparues. La dernière ferme juste le fichier, rien de palpitant. La première, par contre, utilise une petite astuce. On rajoute le caractère défini par le code ASCII 26 (c'est le rôle de la fonction *chr()*) au message. En regardant dans la table, on voit que cela correspond à un caractère EOF (*End Of File* , en théorie la fin d'un fichier, mais bon, ce n'est pas comme ça que se terminent les fichiers, n'ayez crainte…), donc un caractère non imprimable. Il ne va donc, en théorie, **jamais** apparaître dans votre message à dissimuler. À quoi va-t-il donc servir ? En fait, il va tout simplement indiquer la fin du message, et ça va nous être vachement utile pour l'extraction du message…

En petit bonus, je vous montre ce que ça donne quand on a inséré un message. J'ai pris comme exemple un logo de Firefox. La version modifiée peut être vue ici, et l'original à cette adresse.
Apparemment, rien n'a changé, mais si vous regardez cette image avec un éditeur hexadécimal, vous verrez que presque tous les octets auront été modifiés...

La partie Extraction

Une fois qu'on a dissimulé un message dans notre image, on veut aussi pouvoir être capable de retrouver le message initial. La procédure est la suivante :

- on ouvre le fichier BMP ;
- on lit le premier octet ;
- on récupère la valeur des deux bits de poids faible ;
- on convertit cette valeur en base 2 ;
- on ajoute cette valeur binaire à une variable qui va servir de « tampon » ;
- une fois que la variable « tampon » a atteint une taille de 8 caractères (= on a lu 4 octets du fichier bitmap en théorie), on convertit en base 10 et on le convertit en caractère grâce à la table ASCII ;
- si l'on est arrivé au caractère marquant la fin du message, on s'arrête et on affiche le message ;
- on lit le prochain octet ;
- …

Voilà pour la théorie. En pratique, la seule difficulté réside dans la conversion en base 2, et surtout la concaténation avec la variable tampon. Si on laisse faire le PHP tout seul, il va convertir « 0 » en « 0 » et « 1 » en « 1 », alors qu'on aurait préféré avoir du « 00 » et du « 01 ». C'est là où intervient notre fonction *str_pad* qui va de nouveau nous aider. À part ça, rien de bien difficile. En PHP, ça donne la chose suivante :

Code : PHP

```
<?php
$lien = 'images/trucmuche.bmp';
$tampon = "";
$message = "";

$f_image = fopen($lien, 'rb'); // On ne modifie pas le fichier
cette fois-ci, donc le mode r suffit.
fseek($f_image, 54); // On saute le header.
while(!feof($f_image)) // En théorie, on pourrait faire une boucle
infinie que l'on ira « breaker » mais on ne va quand même pas
tenter le diable, hein…

 $octet_image = fread($f_image, 1);
 $octet_image = ord($octet_image); // On lit la valeur du
"charactère" lu.
 $bits_pf     = $octet_image%4;
 $bits_pf     = decbin($bits_pf); // On récupère les deux bits de
poids faible sous forme binaire.
 $bits_pf     = str_pad($bits_pf, 2, '0', STR_PAD_LEFT); // On
ajoute un zéro si nécessaire.
 $tampon      .= $bits_pf; // On rajoute ce qu'on a trouvé au
tampon.

 if(strlen($tampon) == 8)
 {
  // Une fois qu'on a la valeur du caractère du message en base 2.
```

```
        $tampon = bindec($tampon); //conversion en base 10

        if($tampon == 26)
        {
         // Fin du message
         echo $message;
         return;
        }


        $message .= chr($tampon); // Si l'on n'est pas arrivé à la fin du
      message, on ajoute le caractère trouvé et on réinitialise le
      tampon.
        $tampon = "";
       }

      }



      ?>
```

Absolument rien de difficile ici. On a juste traduit en PHP ce qu'on avait énoncé en français tout à l'heure. Évidemment, en C++, ne vous amusez pas à faire autant de *transtypage* qu'ici, mais sinon, à peu de choses près, le principe reste le même.

Sachez qu'il existe une manière plus efficace de procéder, mais pour comprendre comment ça marche il faut être un peu à l'aise avec les opérateur de bits (et surtout être habitué à voir des nombres que comme une série de 1 et de 0 que l'on peut manipuler comme tels). Cette méthode est, à mon avis, préférable lorsque l'on essaie de porter ce programme en C, C++ ou Java, puisqu'il y a moins de transtypage à faire ; de plus, les performances doivent être bien meilleures. Voici donc cette version :

Secret (cliquez pour afficher)

Code : PHP

```
<?php
$lien = 'images/trucmuche.bmp';
$tampon = 0;
$message = "";
$i = 0; //variable qui va servir à compter le nombre d'octets
déjà lus

$f_image = fopen($lien, 'rb'); // On ne modifie pas le fichier
cette fois-ci, donc le mode r suffit.
fseek($f_image, 54); // On saute le header.
while(!feof($f_image)) // En théorie, on pourrait faire une
boucle infinie que l'on ira « breaker » mais on ne va quand même
pas tenter le diable, hein…

 $i++; //très important !

 $octet_image = fread($f_image, 1);
 $octet_image = ord($octet_image); // On lit la valeur du
"charactère" lu.
 $bits_pf     = $octet_image%4;

 $tampon      = ($tampon << 2) | $bits_pf; // On rajoute ce qu'on
a trouvé au tampon.

 if($i % 4 == 0) //c'est-à-dire quand on a lu 4 octets d'affilée
 {
  // Une fois qu'on a la valeur du caractère du message

  if($tampon == 26)
  {
   // Fin du message
   echo $message;
```

```
            return;
          }


          $message .= chr($tampon); // Si l'on n'est pas arrivé à la fin
        du message, on ajoute le caractère trouvé et on réinitialise le
        tampon.
          $tampon = 0;
         }

        }



        ?>
```

Merci à Cortexd pour cette idée.

Voilà, on a déjà terminé… ou presque !

Quelques bonus

Là j'ai juste énoncé un concept, mais on peut bien améliorer le code ! Ce que j'ai également fait, c'est la création d'une nouvelle table de correspondance chiffre <=> lettre, basée sur le ASCII évidemment (donc par exemple, l'apostrophe sera codée par 78 au lieu de 39) qui va servir de « clé ». En gros, on génère une nouvelle table de correspondance à chaque fois que l'on veut chiffrer (il y en a 256! (factorielle 256 c'est-à-dire 1*2*3*4...*256) de différentes… soit à peu près 10^{506}… donc pas mal en fait même si en pratique, ce nombre se laisse réduire à à peu près 10^{60}, ce qui fait tout de même pas mal de clés à tester) et pour déchiffrer, il suffit de connaître cette table de correspondance. Cette clé se présente sous forme d'un fichier dont la position des caractères indique leur nouvelle valeur (par exemple, si le premier octet du fichier-clé vaut 39, alors l'apostrophe sera codée par le chiffre 0, si le deuxième est un octet valant 84, alors le T majuscule sera codé par le chiffre 1, etc.).

Si l'on n'a pas ce fichier-clé, on ne voit qu'un charabia comme message… Mais que l'on peut décrypter sans la clé ! (oui, là on est bien passé dans le domaine de la cryptologie : une fois que le message a été retrouvé dans l'image, c'est son sens qui a été caché). Il s'agit en effet d'un simple codage monoalphabétique que l'on peut casser en analysant la fréquence des lettres, si vous avez chiffré un texte assez long (et si l'attaquant — celui qui a intercepté votre image et veut décrypter ce que vous y avez caché — sait où s'arrête le texte surtout…). En revanche, ça complique la tâche pour quelqu'un analysant toutes les images BMP que vous envoyez et qui essaierait de trouver un nombre anormalement élevé d'assemblages de bits de poids faible donnant un nombre codant pour un caractère imprimable dans la table ASCII…

Le deuxième bonus est l'élargissement de la méthode… à la dissimulation d'un document entier, quel qu'il soit ! Dans ce cas, un petit problème se pose : où s'arrête le document dissimulé ? Il faudrait dans ce cas modifier (par exemple) les 16 premiers octets de l'image pour indiquer la taille de l'image (ça correspond à un entier 32 bits non signé… donc ça permet de stocker des fichiers jusqu'à une taille de 4 Go… largement suffisant). Comme ça, le programme sait où s'arrêter ! Autre chose, ne vous inquiétez pas pour le type de fichier (DOC, ZIP, RAR, JPG, ~~MP3~~). Il est indiqué dans les deux ou trois premiers octets du fichier, en général (à part pour les fichiers texte, bien évidemment). Donc, un système d'exploitation performant (comme GNU/Linux) sait reconnaître le type de fichier automatiquement, sans avoir besoin de l'extension. Ce n'est malheureusement pas le cas de Windows… D'ailleurs, un exercice assez intéressant est associé à cela. On pourrait écrire une fonction qui cache une image dans une autre, et qui permet très facilement de passer d'une image à l'autre. En fait, on va prendre deux images **de même dimension**, et modifier les 4 bits de poids faible d'une image, en les remplaçant par les 4 bits de poids fort de l'autre image. On va considérer que les *headers* vont être les mêmes pour les deux images puisqu'elles ont toutes les deux les mêmes dimensions. Il suffit alors « d'inverser » les octets pour passer d'une image à une autre (en gros, passer de 1101 0011 à 0011 1101). Bien sûr, à la sortie, les deux images vont être quelque peu dégradées, mais elles seront toujours largement reconnaissables… Pour ceux qui voudraient tenter l'aventure, avant de vous lancer dans des trucs compliqués, voici comment arriver à ce résultat en PHP, en trois lignes de code :

Secret (cliquez pour afficher)

Code : PHP

```
<?php
$valeur_octet = 186; //1011 1010 en binaire
```

```
$bits_pf = $valeur_octet % 16; // = 1010 dans cet exemple

$valeur_octet >>= 4; //on décale de 4 vers la droite, ce qui
donne 0000 1011

$nouvel_octet = $valeur_octet + $bits_pf*16; // On ajoute donc
1011 et 1010 0000, ce qui donne bien 1010 1011.

// Les fanatiques des opérateurs de bit et du gain de place
pourront remarquer que cette ligne a exactement le même effet :
// $nouvel_octet = ( ($valeur_octet >> 4) + ($valeur_octet << 4) )
% 256;
// Notez que le "% 256" est inutile dans un langage à typage fort
où l'on peut définir une variable numérique de taille d'un octet,
comme le type byteen Java. Peut-être même que cela marche, sans
le modulo, avec le type unisgned char en C et C++, mais ce n'est
pas sûr…


?>
```

Voilà, juste une dernière petite remarque : le code en tant que tel ne vérifie pas si le message que vous voulez cacher dans l'image est trop grand ou pas. Ce n'est pas très dur à vérifier ; la longueur maximale est donnée par la formule suivante :

longueur_maximale = (taille_image_BMP_en_octets-54)/4

Le 54 correspond à la taille du *header*, et le 4 provient du fait qu'un octet du message est codé dans 4 octets de l'image (2 bits de message par octet d'image). Notez également qu'il faudrait vérifier qu'on écrit bien sur une image BMP, il faut vérifier que les deux premiers octets correspondent à "BM" en majuscules, sans les guillemets bien entendu.

Très important aussi (comme l'a fait remarquer Jet74) : la compression utilisée dans le format PNG est non destructive. C'est-à-dire que vous pouvez convertir votre image modifiée au format PNG, l'envoyer au destinataire, qui lui n'aura qu'à la reconvertir au format BMP et extraire le texte normalement. Ainsi, on obtient une image moins lourde et surtout moins suspecte ! (le format BMP n'étant plus tellement utilisée que ça, toute image BMP trainant dans les parages peut paraître suspecte...)

Et voilà, je crois que c'est tout !
Ainsi, vous avez vu à quel point il est simple de dissimuler du texte —mais aussi n'importe quel autre type de document au final— dans une image BMP, sans modifier son apparence. Malheureusement, cette méthode peut être utilisée pour des fins très diverses qui ne sont pas toujours louables —du terrorisme à la pédopornographie, en théorie. Cela dit, je vous fais assez confiance pour ne pas en faire n'importe quoi ; et au pire, cacher quelques mots doux dans un smiley en forme de cœur.