

# Qu'est-ce qu'un éditeur hexadécimal ?

Un éditeur hexadécimal est un logiciel permettant de visualiser et éditer le contenu de toutes sortes de fichier, en visualisant leurs données en octets présentés sous forme de caractères hexadécimaux. L'hexadécimal est le fait de compter en base 16, qui est très utilisé en informatique car facile à convertir en binaire, et inversement. On y dénombre donc 16 caractères différents, de 0 à F, soit : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E et F. En informatique, un octet est en effet couramment représenté par deux de ces caractères. Voici quelques exemples de tels éditeurs : **Free Hex Editor**, **HexEdit**, **HxD**, **WinHex**.

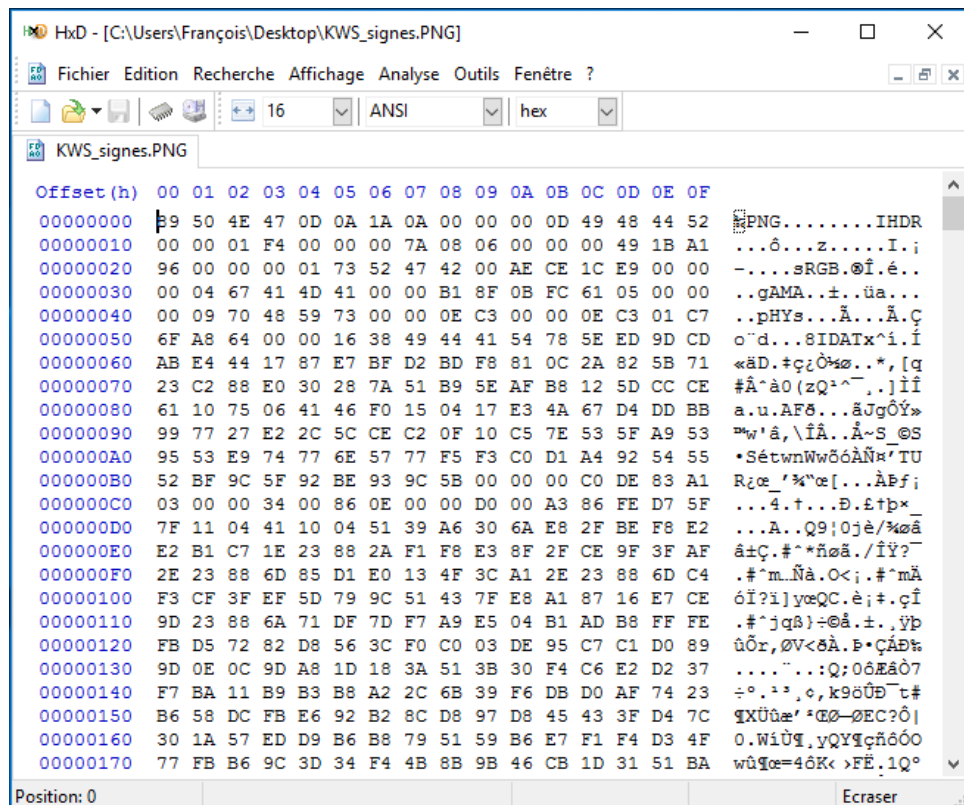
## Des fichiers binaires ?

Par définition, un fichier binaire est un fichier qui n'est pas interprétable sous forme de texte. Par exemple : une image, une piste audio, une vidéo, etc ...

Le terme peut aussi désigner un fichier "exécutable", autrement dit un programme obtenu à partir de la compilation du code source d'un langage de programmation. Sous Windows, ces fichiers se présentent typiquement avec l'extension **.exe** ou **.msi**, même si il peut en exister d'autres. Ils seront générés par un compilateurs, depuis des langages tels que **Java**, **C++**, **C#** ou encore **Visual Basic** pour ne citer que les plus connus.

A l'aide d'un éditeur hexadécimal, il sera possible de consulter le contenu intégral de n'importe quel fichier. Le mot intégral a son importance ici : il sera ainsi possible d'accéder à TOUTES les données d'un fichier, y compris les métadonnées, alors que la plupart du temps ce n'est pas le cas lorsque le fichier est ouvert avec un logiciel spécialisé. Par exemple, lors de l'ouverture d'une photo par un logiciel de traitement d'image, l'image sera affichée alors que les données permettant au logiciel de l'interpréter correctement (comme la taille, le format, les dimensions, le taux de compression, etc ...) seront bien souvent masquées.

Par défaut, un éditeur hexadécimal présente les données par ligne de 16 octets chacuns notés sur 2 chiffres hexadécimaux, suivis sur la même ligne par l'interprétation de ces données sous forme de texte. Ci-dessous un aperçu du logiciel **HxD** (avec lequel nous avons ouvert une image au format **.png**) qui est l'éditeur hexadécimal que nous utiliserons tout au long de ce document :



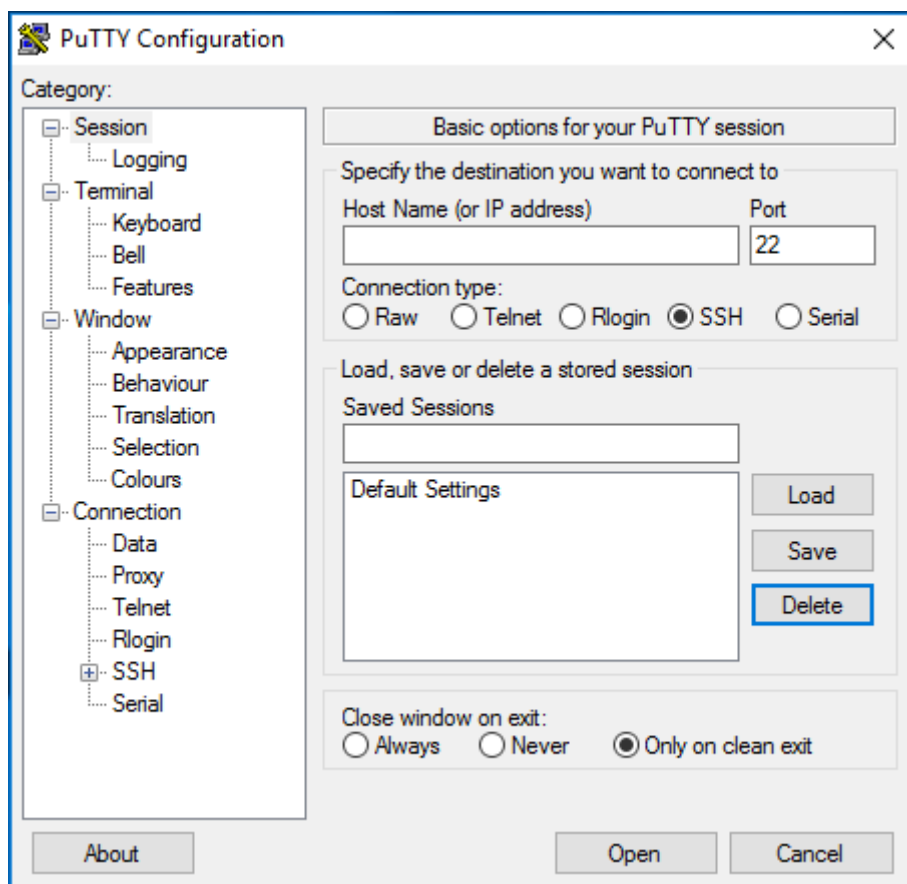
Nous allons voir plusieurs exemples d'utilisation concrète de modification de fichiers à l'aide d'un tel éditeur. Nous utiliserons donc HxD sur Windows 10, mais vous pouvez bien sûr en utiliser un autre si vous le préférez ou si vous travaillez sur un autre système d'exploitation. Il existe de nombreuses possibilités, mais nous ne verrons ensemble que quelques exemples particuliers.

**Attention** : comme nous le verrons plus en détail par la suite, notre marge de manœuvre quant à l'édition des données des fichiers sera limitée. En effet, à défaut de disposer d'un programme spécialisé pour modifier notre fichier cible (un logiciel de traitement d'image pour une photo, un logiciel d'édition audio pour une piste sonore, ou un compilateur dans le cas d'un fichier exécutable avec l'extension .exe) nous serons contraints de respecter au maximum sa structure. Pour parler plus concrètement, on ne pourra remplacer une portion d'octets que par une autre portion de même taille ou plus petite au sein d'un fichier, sans quoi il pourrait devenir inutilisable : nous verrons des exemples illustrant précisément cette limitation plus bas dans ce document ...

## Exemples d'utilisations concrètes

### Traduction d'un programme

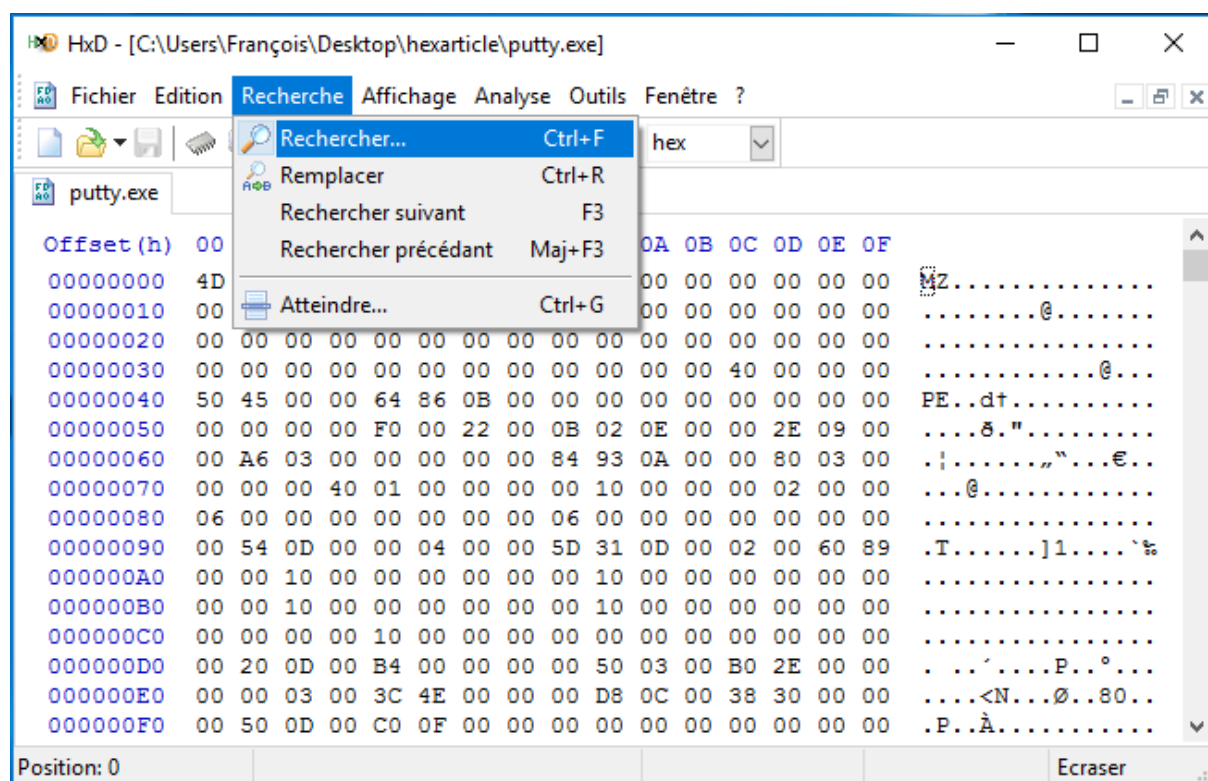
Prenons pour l'exemple le logiciel **PuTTY** : bien connu dans le milieu GNU/Linux, il permet notamment d'établir des connexions en SSH. Dans notre cas nous n'utiliserons que le **client SSH/Telnet** en lui-même, téléchargez donc sa dernière version pour Windows, et vous obtiendrez un programme comme ceci :



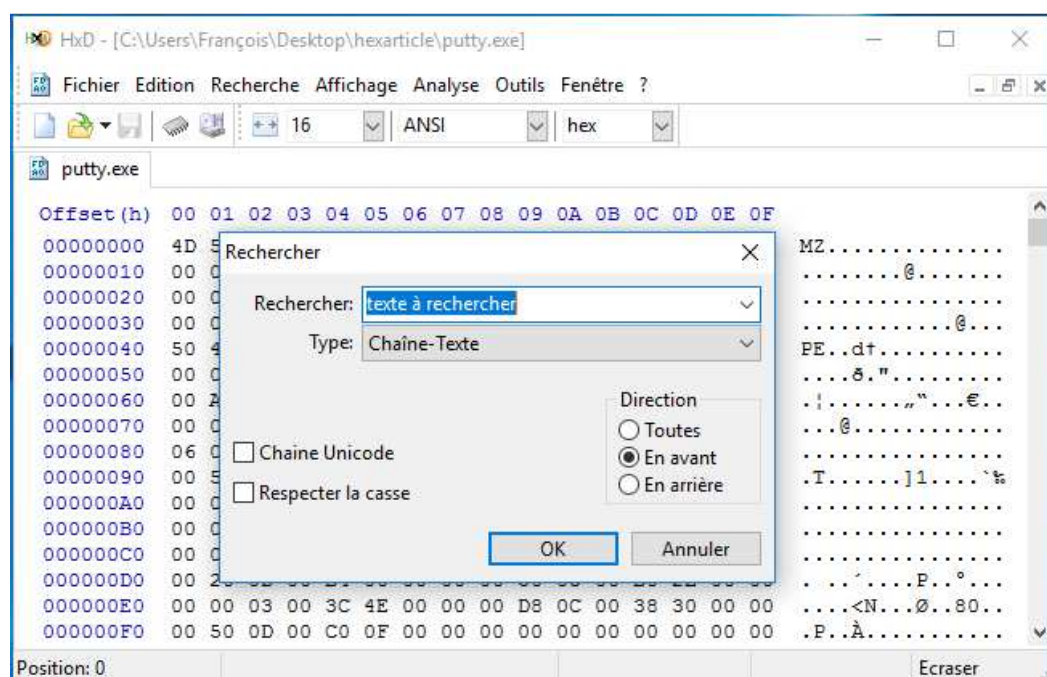
Ce client se présente sous la forme d'un unique fichier exécutable (**putty.exe** est le seul fichier nécessaire à son utilisation) et existe uniquement en langue anglaise. Nous allons essayer de le traduire (au moins partiellement) en modifiant ce fichier exécutable : pour cela, tentons de localiser, au sein des données affichées par notre éditeur hexadécimal, des portions de texte visibles dans l'interface graphique du logiciel.

**Important : par sécurité, n'oubliez pas d'effectuer une sauvegarde du fichier avant toute manipulation !**

Ouvrons donc le fichier **putty.exe** avec HxD pour visualiser son contenu. Effectuons ensuite une recherche pour localiser du texte :

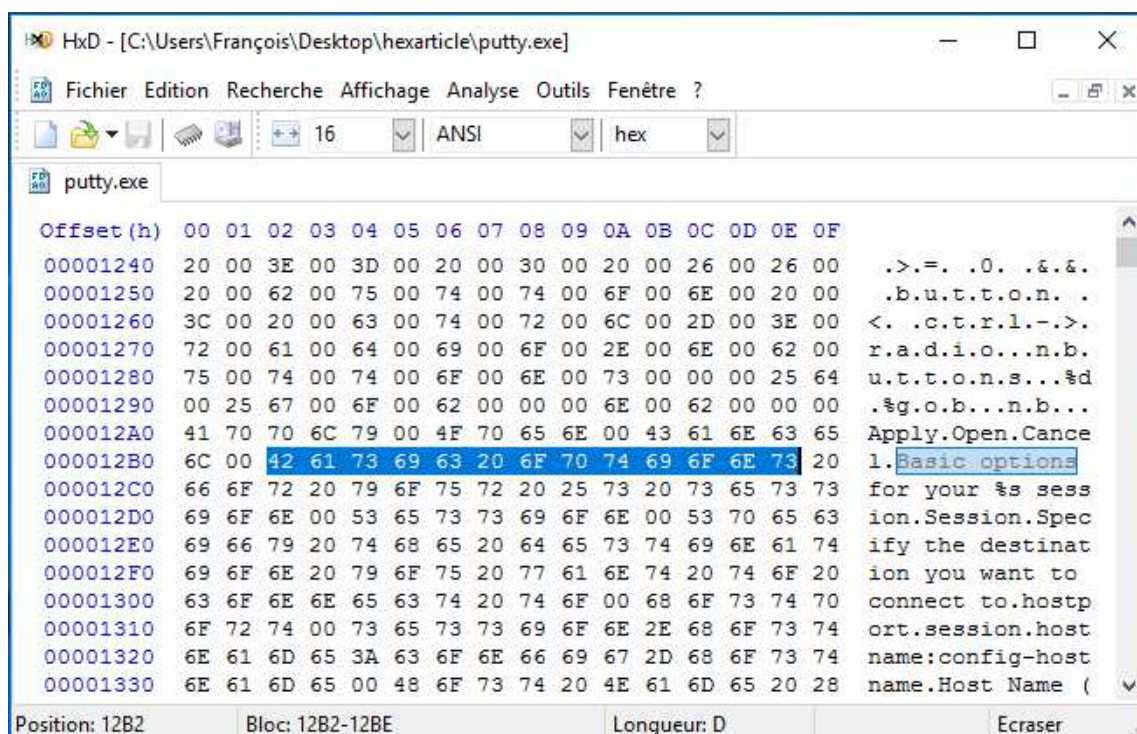


Dans le menu de recherche, choisissez l'option "**Chaîne-Texte**" et renseignez le texte à rechercher, comme montré dans l'image suivante. Selon l'encodage du texte qu'on voudra trouver, on pourra cocher la case "**Chaîne Unicode**" même si dans le cas de PuTTY ce ne sera pas nécessaire car il utilise à priori l'encodage **ANSI** (l'encodage par défaut de Windows). Si plusieurs correspondances sont trouvées pour une recherche, vous pouvez utiliser F3 (par défaut) comme raccourci pour naviguer d'un résultat au suivant :

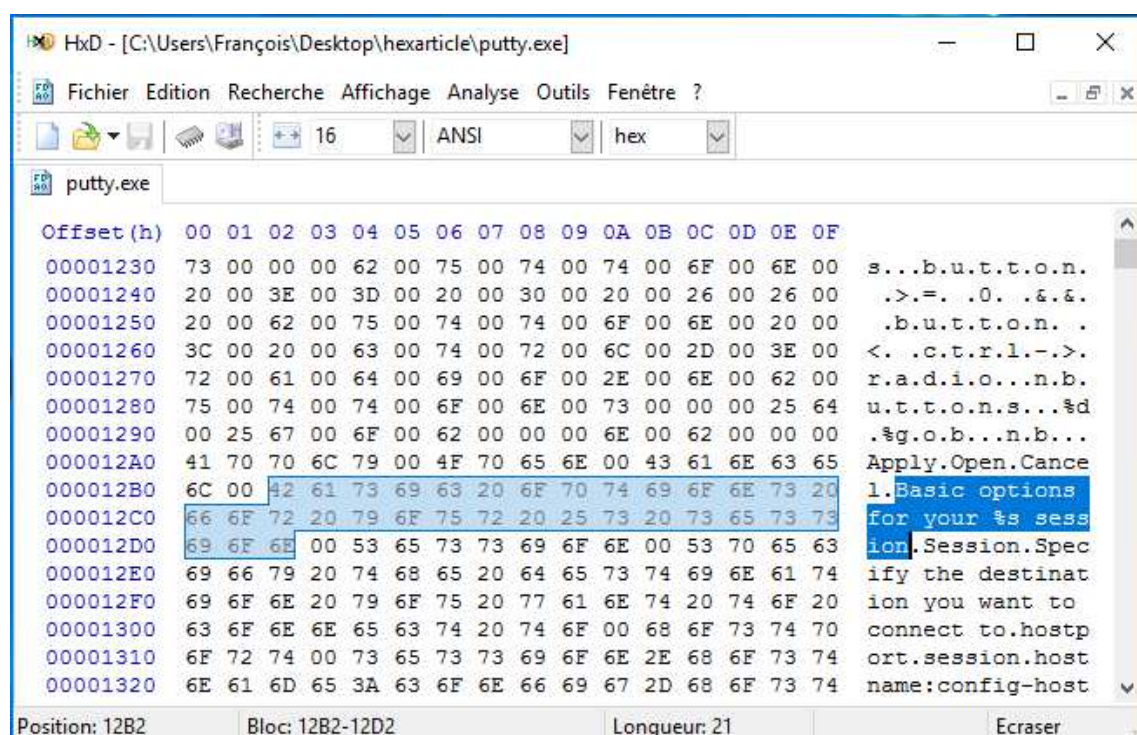




Par exemple, recherchons la chaîne "Basic options" provenant du texte "Basic options for your PuTTY session" situé en titre de la partie supérieure droite du logiciel :



Sur la capture ci-dessus, nous pouvons constater que la chaîne recherchée commence à la position 12B2 et termine à la position 12BE : voir le coin inférieur gauche de HxD. A noter que **position** est parfois aussi appelé **offset**. Étendons maintenant notre sélection d'octets afin de couvrir tout le texte ciblé :

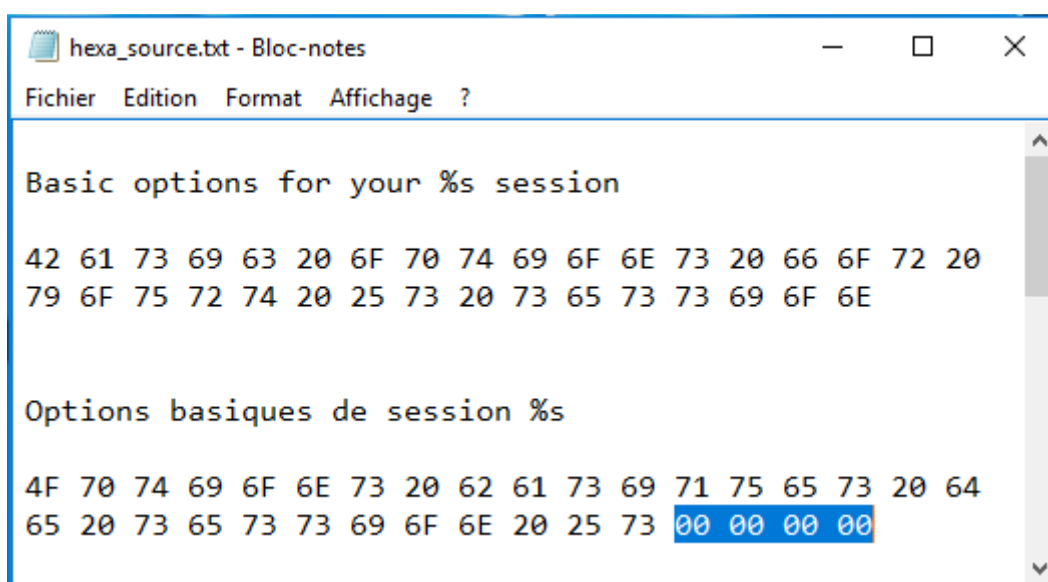


Maintenant la prochaine opération consistera à utiliser un convertisseur de texte vers hexadécimal (vous pouvez par exemple utiliser un site comme <http://string-functions.com/string-hex.aspx>) pour transformer notre chaîne de caractères en un ensemble de valeurs hexadécimales que nous insérerons dans le fichier avec HxD.

**Attention : comme dit précédemment, il est important de ne pas insérer plus de valeurs que leur nombre d'origine, sinon le fichier sera corrompu.**

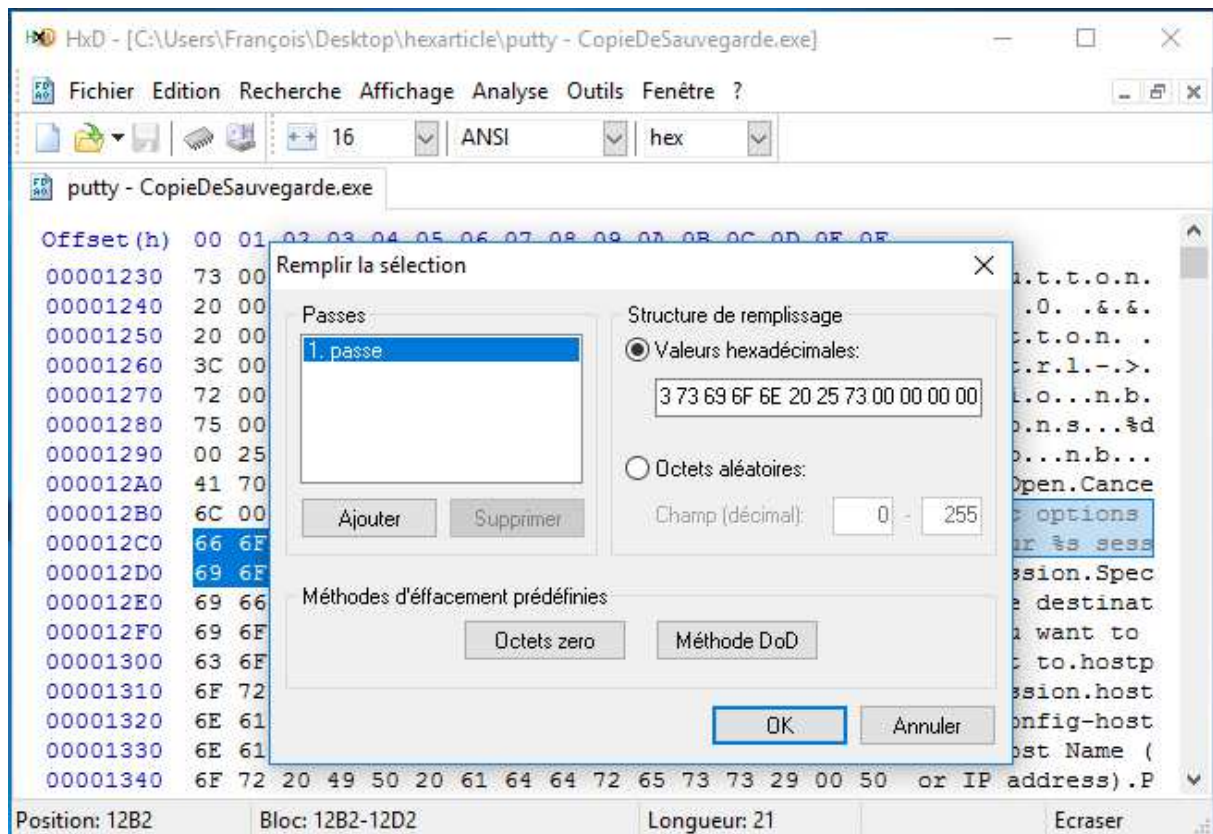
Concrètement, sur notre exemple, les valeurs que nous insérons ne devront donc pas dépasser la position **12D2**.

Ce ne sera pas un problème car nous choisirons le texte de substitution suivant, plus court que l'original : "**Options basiques de session %s**". Voir ci-dessous la comparaison entre le texte et son équivalent en valeurs hexadécimales.

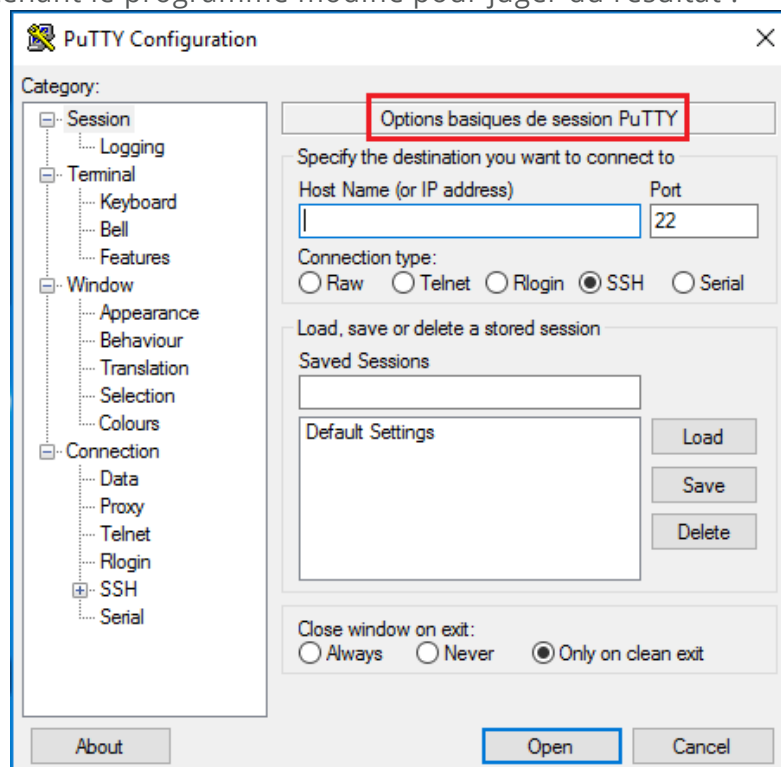


Justement, afin d'être sûr que des caractères de l'ancien texte "**Basic options for your %s session**" ne restent pas affichés après le remplacement, nous allons les "effacer" en donnant comme valeur **00** aux quatre derniers octets. Ainsi les deux chaînes de caractères seront de même longueur, et le nombre d'octets à remplacer sera le même.

Pour effectuer le remplacement de valeurs sur HxD : Surlignez les octets à remplacer (comme dans l'avant-dernière capture), puis effectuez un **clic droit** -> cliquez sur **Remplir la sélection** -> choisir **Structure de remplissage : Valeurs hexadécimales** et insérer les valeurs dans le champ, comme montré ci-dessous :



Une fois leur valeur changée, les octets concernés seront affichés en évidence, en rouge. Validez en enregistrant les modifications (en cliquant sur **Fichier -> Enregistrer** ou avec le raccourci **Ctrl + S**). A ce moment-là, une copie du fichier dans sa version originale devrait être créée par mesure de sécurité, avec l'extension **.bak**.  
Lançons maintenant le programme modifié pour juger du résultat :





Comme le montre la capture ci-dessus, le texte a bien été remplacé dans le programme. Ce sera tout pour ce premier exemple. La méthode montrée est certes fastidieuse mais a le mérite d'exister et pourrait s'avérer utile pour quelqu'un de (très) motivé, et ne disposant pas du code source du logiciel.

## Modification d'un chemin d'accès dans un programme

Nous allons à présent nous intéresser à une autre possibilité : la modification du chemin d'accès vers un fichier dans le code d'un programme. Pour écrire ce code, nous nous servons du **C#**, un langage de programmation que j'apprécie particulièrement. Ainsi seront montrées ci-dessous des captures d'écrans de **Visual Studio** (Entreprise 2017), l'environnement de développement intégré phare de Microsoft.

Prenons l'exemple du programme suivant, qui va tenter d'afficher le contenu d'un fichier dont l'emplacement est bien spécifique :

```
1  using System;
2  using System.IO;
3
4  namespace TestPathHexEdition {
5      class Program {
6          static void Main(string[] args) {
7
8              /* Exemples de chemins d'accès :
9              *
10             * C:\chemin\vers\le\fichier\text.txt
11             * C:\chemin\alternatif\vers\text.txt
12             * C:\chemin\alternatif\vers\data.csv
13             *
14             */
15
16             try {
17                 using (StreamReader reader = new StreamReader(@"C:\chemin\vers\le\fichier\text.txt")) {
18
19                     string line;
20
21                     Console.Write("Contenu : \n");
22                     Console.ForegroundColor = ConsoleColor.Yellow;
23
24                     while ((line = reader.ReadLine()) != null) {
25                         Console.WriteLine(line);
26                     }
27                 }
28             }
29             catch {
30                 Console.ForegroundColor = ConsoleColor.Red;
31                 Console.WriteLine("Erreur lors de la lecture du fichier !");
32             }
33
34             Console.ReadKey(); // pause
35         }
36     }
37 }
38
39
```

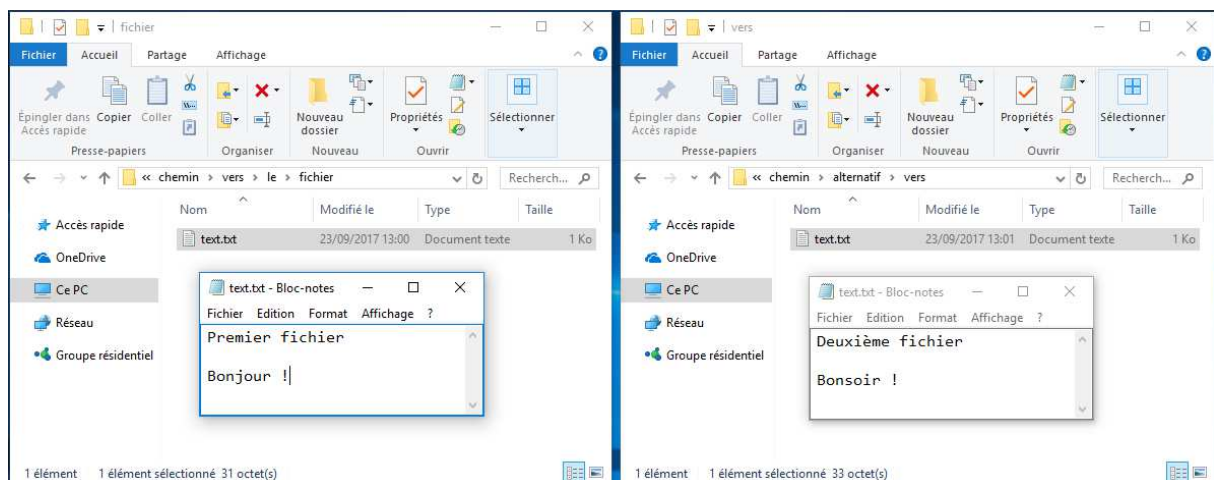
Il s'agit d'un programme de type console dans lequel nous créons un objet **reader**, dérivé de la classe **StreamReader**, auquel nous fournissons en paramètre le chemin



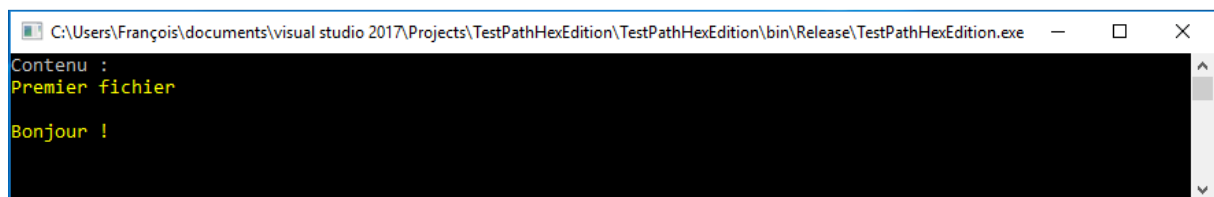
d'accès absolu vers le fichier à lire. Dans notre cas, il s'agit de "C:\chemin\vers\le\fichier\text.txt". Pour détailler le reste du code, notons que les blocs d'instruction **try** et **catch** nous permettent de nous assurer que le programme ne rencontre pas une exception inattendue si le fichier n'est pas trouvé, ou s'il n'est pas lisible. Si jamais c'était le cas, le programme ne quitterait pas inopinément mais afficherait une erreur à l'utilisateur, à l'aide d'un texte coloré en rouge dans la console.

Le mot clé **using** nous permet quant à lui de n'utiliser l'objet **reader** que pendant le laps de temps où il nous est utile : concrètement, il sera automatiquement détruit une fois la lecture du fichier terminée. Dernier point, la boucle "**while((line = reader.ReadLine()) != null){**" (traduisible littéralement par "**tant que la ligne lue par reader n'est pas inexistante, faire ...**") incluse à l'intérieur permet d'effectuer un traitement pour chaque ligne lue dans le fichier : en l'occurrence nous afficherons chacune de ces lignes en jaune dans la console.

Voici ci-dessous un aperçu de deux fichiers : celui auquel le programme fait référence, et un autre dont le chemin d'accès complet est "C:\chemin\alternatif\vers\text.txt". Les deux chemins d'accès sont de longueur équivalente, ce qui ne manquera pas de nous faciliter les choses.

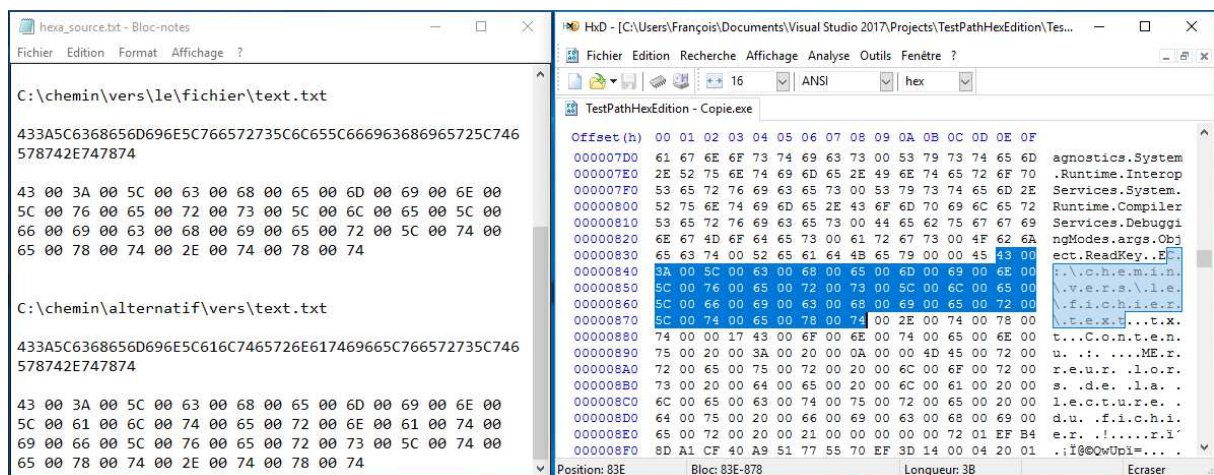


Maintenant, voyons le résultat obtenu lors de l'exécution du programme :

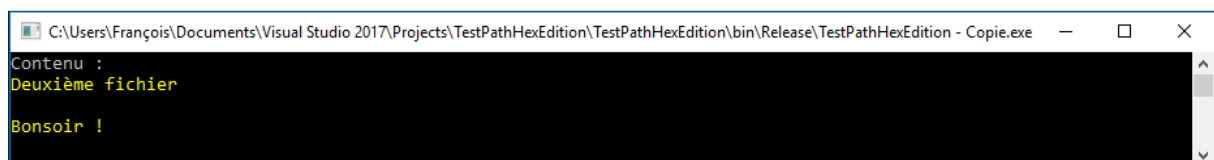


Maintenant recherchons et modifions le chemin d'accès "C:\chemin\vers\le\fichier\text.txt" en "C:\chemin\alternatif\vers\text.txt" afin de faire

lire le deuxième fichier au programme. Pour ce faire, utilisons l'éditeur HxD et, comme dans le premier exemple avec putty.exe, effectuons une recherche de caractères (comme par exemple "fichier") dans le fichier exécutable de notre programme (dans mon cas **TestPathHexEdition.exe**) en prenant soin de cocher l'option "**Chaîne Unicode**" (UTF-8) car il s'agit de l'encodage par défaut utilisé par Visual Studio. Ensuite : étendre la sélection d'octets pour englober tout le chemin d'accès, puis effectuer un **Clic droit** -> cliquer sur **Remplir la sélection** -> Choisir **Valeur hexadécimale** et insérer les valeurs correspondant au deuxième chemin d'accès dans le champ prévu à cet effet :



Notez cependant les nombreux octets dont la valeur est **00**, détail qui a son importance lorsque nous travaillons avec des chemins d'accès. Ainsi lorsque vous convertissez "C:\chemin\alternatif\vers\text.txt" en valeurs hexadécimales, pensez à insérer ensuite des **00** entre chaque octet. N'hésitez pas à utiliser des notes pour mieux vous y retrouver ! Voir la capture ci-dessus (à gauche) pour une comparaison entre trois écritures différentes pour chaque chemin d'accès : 1) écrit normalement 2) écrit en hexadécimal 3) prêt à être inséré avec HxD. Voici le résultat après édition de l'exécutable (dans mon cas, de l'édition d'une copie de l'exécutable) :



Nous pouvons constater que programme charge bien le contenu du deuxième fichier. Remarque : j'ai choisi l'encodage **UTF-8** pour enregistrer mes fichiers texte, et non **ANSI** (proposé par défaut) ce qui permet d'afficher normalement le caractère "è" contenu dans "Deuxième fichier".

C'est la fin de ce deuxième exemple. Encore une fois, l'utilité de telles opérations peut ne pas paraître évidente, d'autant plus qu'ici nous nous sommes contentés de changer un chemin d'accès simpliste pour un autre. Difficile de faire beaucoup mieux sans le code source du programme.

Pour avoir un exemple d'utilisation réellement utile, je vous invite à regarder **DS-ExeModifier** sur GitHub, un programme dont le but est de modifier l'exécutable d'un jeu vidéo (par exemple Dark Souls) afin de charger des mods, c'est à dire du contenu alternatif comme par exemple des textes ou des sons/musiques. Il s'agit d'un programme écrit en C# qui se présente sous la forme d'une interface graphique intuitive, dont l'objectif est de permettre à l'utilisateur de changer l'emplacement depuis lequel les données de jeu sont chargées, sans avoir besoin d'utiliser un éditeur hexadécimal (avec lequel on effectuerait les opérations décrites plus haut).

## Écrire des valeurs hexadécimales avec la programmation

Nous allons maintenant détailler le fonctionnement d'un programme qui va écrire des octets dans un fichier binaire, toujours en utilisant le **C#** (et **Visual Studio** Entreprise 2017).

Il s'agit d'un programme de type **console** avec lequel nous allons de nouveau interagir avec le fichier `putty.exe` pour remplacer le texte **"Basic options for your %s session"**, comme dans le premier exemple vu plus haut. Le programme est divisé en six captures d'écran que nous détaillons ci-dessous :

```
1  using System;
2  using System.IO;
3  using System.Linq;
4  using System.Text;
5
6  namespace HexEditPJT {
7      class Program {
8          static void Main(string[] args) {
9
10
11             // ----- Start-up checks -----
12
13             if (!File.Exists("putty.exe")) {
14                 Console.WriteLine("File \"putty.exe\" not found ! Press any key to exit ...");
15                 Console.ReadKey();
16                 Environment.Exit(0);
17             }
18
19             if (!File.Exists("putty.exe.bak")) {
20                 File.Copy("putty.exe", "putty.exe.bak");
21             }
22         }
23     }
24 }
```

Dans un premier temps, à partir de la ligne 11, nous vérifions que le fichier **putty.exe** existe bien dans le répertoire courant. Si c'est effectivement le cas, nous créons une copie de sauvegarde de celui-ci (si ce n'est pas déjà fait) par sécurité. Par contre si le fichier est introuvable, un texte d'erreur est écrit dans la console et le programme se termine après la confirmation de l'utilisateur.

```
23
24 // ----- Variable declarations -----
25
26 int startPos = 0x12B2; // 0x12B2 (hex) = 4786 (decimal)
27
28 int byteCount = 0;
29 string userInput = "";
30 string bytesToAdd = "";
31 bool userInputOk = false;
32
33 byte[] bytesToWrite;
34 byte[] lastBytes;
35
```

Une fois les vérifications terminées, nous déclarons l'ensemble des variables utilisées dans la suite du programme et notamment **startPos** qui correspond à la position où débute l'écriture du texte "**Basic options for your %s session**" dans **putty.exe**, soit **12B2** : notez qu'en C# une valeur en hexadécimal est précédée par le préfixe **0x** et est alors considérée comme un entier codé sur 32 bits (**Int32**), en tant que variable.

```
36
37 // ----- Reading bytes from file -----
38
39 using (BinaryReader reader = new BinaryReader(File.OpenRead("putty.exe"))) {
40     try {
41         reader.BaseStream.Position = startPos;
42
43         while (BitConverter.ToString(reader.ReadBytes(1)) != "00") {
44             byteCount += 1;
45         }
46     } catch {
47         Console.WriteLine("Error while reading file");
48     }
49 }
50
51
52 Console.WriteLine("Max length : " + byteCount);
53
```

Ici nous reprenons la structure du code vu plus haut dans l'article, avec l'instruction **using** dans laquelle nous créons **reader**, un objet dérivé de la classe **BinaryReader** auquel nous fournissons en paramètre le chemin du fichier avec lequel il doit interagir, soit **putty.exe**, et ce en mode **Ouverture-Lecture**. La propriété **Position** permettra comme son nom l'indique de se positionner précisément dans le fichier : c'est ce qui marquera notre point de départ pour les opérations de lecture.

Ensuite, nous utilisons une boucle **while** pour étudier chaque octet à partir de cette position : tant que la valeur hexadécimale de l'octet lu n'est pas **"00"**, la valeur



de **byteCount** sera incrémenté de 1 et le lecteur avance à la position suivante (c'est à dire l'octet suivant). Dans ce contexte, cela nous permet de compter (et d'afficher dans le programme : voir la ligne 52) le nombre maximum de caractère insérables pour remplacer notre texte dans PuTTY : dans notre cas ce maximum sera de 33 comme nous le verrons plus bas.

```
54
55 // ----- Choosing the string to insert -----
56
57 while (userInputOk == false) {
58     Console.WriteLine("\nString to insert :");
59     userInput = Console.ReadLine();
60
61     if (userInput.Length > 0 && userInput.Length <= byteCount) {
62         userInputOk = true;
63     }
64     else {
65         Console.WriteLine("Error : incorrect string length ! Press any key to try again ...");
66         Console.ReadKey();
67     }
68 }
69
70 bytesToWrite = Encoding.Default.GetBytes(userInput);
71
72
73 // ----- Adding bytes to match "byteCount" length, if necessary -----
74
75 for (int i = userInput.Length; i < byteCount; i++) {
76     bytesToAdd += "00";
77 }
78
79 lastBytes = StringToByteArray(bytesToAdd);
80
```

Dans cette capture, nous demandons à l'utilisateur de renseigner le texte qu'il souhaite insérer. Tant que la longueur de sa saisie est de 0, ou si elle est supérieure à la valeur de **byteCount** calculé plus haut, une erreur apparaît et il devra à nouveau saisir du texte. Une fois que la saisie est validée, nous obtenons son équivalent en une suite d'octets grâce à la fonction **GetBytes** de la ligne 70.

Ensuite, nous nous servons d'une boucle **for** pour (si cela est nécessaire) alimenter la variable **bytesToAdd**, afin que finalement le nombre d'octets que nous écrirons à la prochaine étape soit égal à 33, soit la valeur de **byteCount** (donc pour rappel la longueur du texte "**Basic options for your %s session**"). Le but de cette opération est de s'assurer qu'il ne subsiste pas des caractères de l'ancien texte après remplacement des valeurs dans le fichier : en d'autres termes, faire en sorte que les valeurs hexadécimales correspondant respectivement à l'ancien et au nouveau texte soient de même longueur. Enfin, la fonction **StringToByteArray** utilisée en dernière ligne permet de convertir la valeur de **bytesToAdd** en une suite d'octets.

```

81
82 // ----- Writing bytes to file -----
83
84 using (BinaryWriter writer = new BinaryWriter(File.OpenWrite("putty.exe"))) {
85     try {
86         writer.BaseStream.Position = startPos;
87         writer.Write(bytesToWrite);
88         writer.Write(lastBytes);
89     }
90     catch {
91         Console.WriteLine("Error while writing in file");
92     }
93 }
94
95 Console.WriteLine("\nSuccess ! Press any key to exit ...");
96 Console.ReadKey();
97
98
99

```

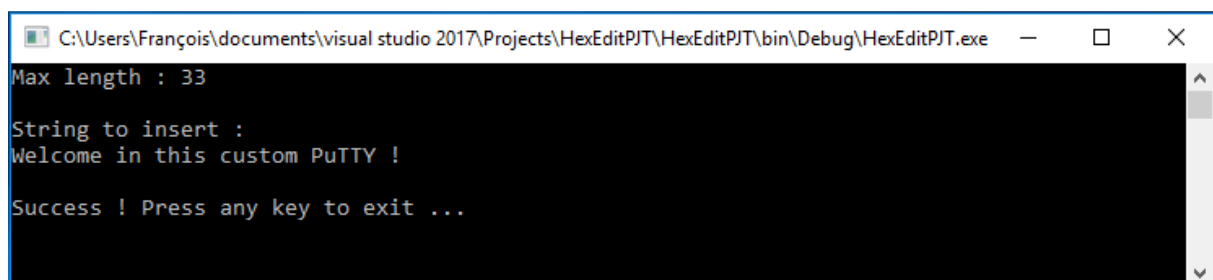
Ci-dessus, nous procédons justement aux opérations d'écriture dans le fichier : notez cette fois que c'est le mode **Ouverture-Écriture** qui est spécifié dans la construction de l'objet **writer** (classe **BinaryWriter**). Comme lors de l'utilisation de l'objet **reader** plus haut dans le programme, nous utilisons la propriété **Position** pour nous placer à l'endroit où débute le texte à remplacer dans le fichier. Une fois l'écriture terminée (et réussie) à l'aide de la fonction **Write**, le programme se termine.

```

100
101 // https://stackoverflow.com/questions/321370/how-can-i-convert-a-hex-string-to-a-byte-array
102 1 référence
103 public static byte[] StringToByteArray(string hex) {
104     return Enumerable.Range(0, hex.Length)
105         .Where(x => x % 2 == 0)
106         .Select(x => Convert.ToByte(hex.Substring(x, 2), 16))
107         .ToArray();
108 }
109
110

```

Ci-dessus : la fonction **StringToByteArray** retournant une suite d'octets. Elle provient du site Stack Overflow, comme indiqué dans le commentaire la précédant, et utilise le composant .NET LINQ (qui propose une syntaxe proche de celle du langage SQL) que nous avons importé dans les premières lignes du programme.

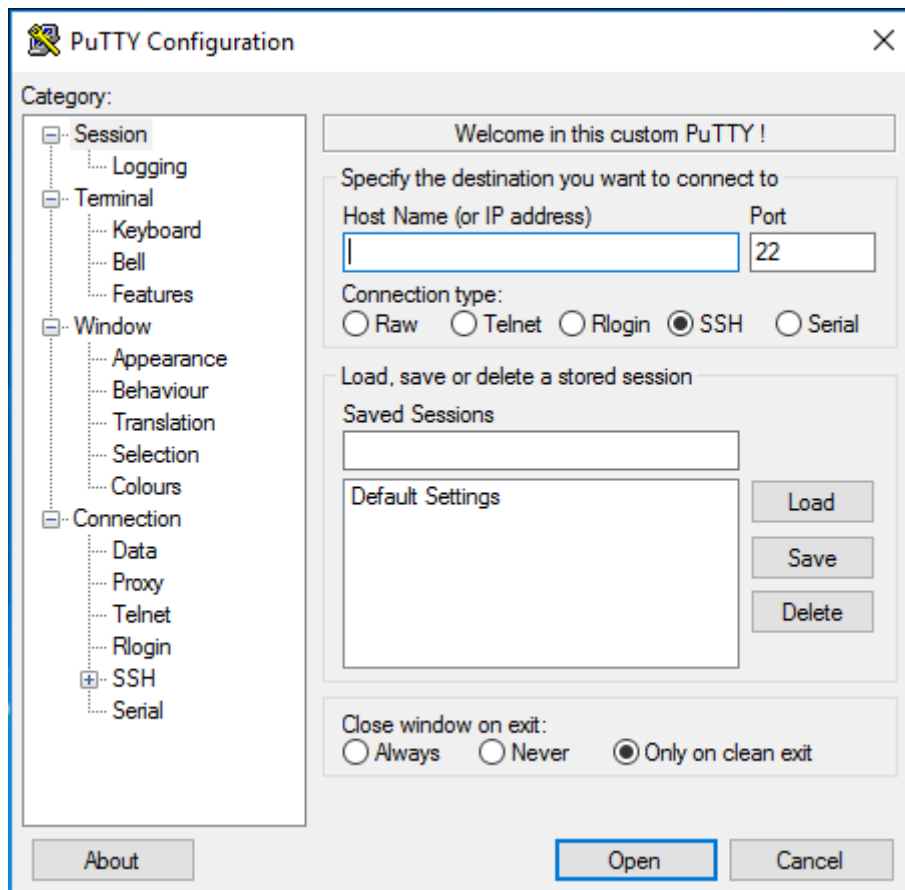


```

C:\Users\François\documents\visual studio 2017\Projects\HexEditPJT\HexEditPJT\bin\Debug\HexEditPJT.exe
Max length : 33
String to insert :
Welcome in this custom PuTTY !
Success ! Press any key to exit ...

```

Ci-dessus, voici le résultat de l'exécution du programme. Comme vous pouvez le voir, nous sommes limités à 33 caractères : le texte choisi pour être inséré est "**Welcome in this custom PuTTY !**". Il ne nous reste plus qu'à lancer le programme pour vérifier que l'insertion a bien fonctionné :



C'est bien le cas visiblement ! On constate que malgré le fait que le nouveau texte inséré soit plus court, il reste centré à l'affichage et ne contient pas de caractères provenant de l'ancien. Remarque : l'encodage des caractères utilisé par le logiciel est également celui utilisé par défaut sous Windows, à savoir ANSI. C'est pourquoi nous avons bien précisé "**Encoding.Default.GetBytes**" dans notre code, à la ligne 70, là où **Default** aurait été remplacé par **UTF8** dans bien d'autres cas.

Voilà, c'est la fin de cet exemple de programme. Bien sûr il reste une certaine marge d'amélioration : par exemple, et bien qu'il y ait au final peu de répétitions dans le code, il serait sans doute plus élégant d'utiliser des fonctions pour mieux le segmenter. Aussi il ne s'agit que d'une utilisation très spécifique et peu réaliste, où nous ciblons précisément une certaine portion d'octets à éditer. En l'état, il s'agit tout de même d'une démonstration concrète de l'édition d'un fichier en C#, dont les fonctionnalités intégrées (notamment les classes **binaryReader** et **binaryWriter** et leurs méthodes) nous facilitent bien la tâche.

## Conclusion

L'édition de fichiers grâce à l'hexadécimal est un vaste sujet. Dans cet article, nous n'avons fait qu'effleurer les possibilités offertes par de tels outils, en ne manipulant

notamment que des chaînes de caractères pour remplacer des portions de texte localisées.

Mais au-delà de ces exemples simples existent de nombreuses autres applications dont certaines sont tout de même assez complexes. On pense rapidement à l'ingénierie inversée (**reverse engineering** en anglais), dans laquelle on peut être amené à utiliser de telles solutions mais aussi à des cas plus concrets, comme par exemple l'édition de sous-titres incrustés dans une vidéo pour y corriger une faute. Malheureusement cet exemple, de par les nombreux formats vidéo et méthodes d'incrustation de sous-titres existants, n'est pas assez reproductible pour qu'il soit réellement intéressant de l'aborder ici.