# Maldev for dummies

## Exercice 1 : Shellcode Loader

### Explication des fonctions :

*VirtualAlloc :*

a. Réserve une zone de mémoire virtuelle
b. Réserve de la mémoire RWE (Read Write Execute)

- static extern IntPtr VirtualAlloc(
  IntPtr lpAdress, ← Adresse mémoire souhaitée. [IntPtr.Zero ← Windows choisit lui-même]
  UIntPtr dwSize, ←Taille de la mémoire à allouer en octet
  UInt flAllocationType, ← Comment allouer [MEM_COMMIT = 0x1000 & MEM_RESERVE = 0x2000] (Si COMMIT && RESERVE = 0x3000)
  Uint flProtect) ; ← Droit mémoire (RWE). RW → 0x04 || ER → 0x20 || RWE → 0x40

  ⚠️ Pas de gestion du Garbage collector, on peut oublier de libérer la mémoire (Fuite de mémoire) ou on peut donner les mauvais droits et ça peut faire crasher le programme ⚠️
  Bonne pratique : VirtualFree(ptr,UInt Ptr.zero, 0x8000) ; ← Libère la mémoire

*CreateThread :*

- static extern IntPtr CreateThread(
  IntPtr lpThreadAttributes, ← Sécurité du Thread. [Int.Zero ← Valeur par défaut]
  UIntPtr dwStackSize, ←Taille de la pile. [Int.Zero ← Valeur par défaut]
  IntPtr lpStartAdress, ←Adresse de la fonction exec par le thread
  Intptr lpParameter, ←Paramètre passé à la fonction du thread
  uint dwCreationFlags, ← si 0, démarre immédiatement. Si 0x4 ← Démarre en pause
  out uint lpThreadIp) ; ← Reçoit l'ID du Thread.

*WaitForSingleObject :*

- static extern uint WaitForSingleObject(
  IntPtr hHandle, ← handle à attendre (Comme par exemple un CreateThread)
  uint dwMilliseconds) ; ←Temps d'attente (oxFFFF…) Attente indéfinit
  - Retour :
    - WAIT_OBJECT_0 : Object terminé
    - WAIT_TIMEOUT : Temps écoulé
    - WAIT_FAILED : Erreur

# Exercice 2 : ShellCode Injector

## Explications basique de certains paramètres :

[Flag] est un attribut qui indique que des valeurs d'un enum peuvent être combinées entres elles.

Enum :
[Flags]

```
public enum ProcessAccessFlags : uint
{
   All = 0x001F0FFF
}
```

Au lieu d'écrire plus tard, 0x001F0FFF on écrira simplement ProcessAccessFlags.All ← ça évite les erreurs et si une erreur est faite dans 0x... on a 1 seul endroit a changer pour tout le code.

All = 0x001F0FFF est un masque binaire qui combine PROCESS_VM_READ, ..._WRITE,...

## Explication des fonctions :

## OpenProcess :

a. Une fonction WinAPI qui sert à obtenir un handle (identifiant que Windows donne pour interagir avec un processus) vers un processus déjà en cours d'exécution.
   - public static extern IntPtr OpenProcess(
     ProcessAccessFlags dwDesiredAccess, ← Droits demandés sur le processus. ⚠️ Windows refuse si trop de droits sans privilèges suffisants.
     bool bInheritHandle, ← Indique si le handle peut-être hérité par un processus enfant.
     int dwProcessId); ← PID du processus cible. [explorer.exe : PID 1234, notepad.exe : PID 5678]
     - Retour :
       - IntPtr handle != IntPtr.Zero ← ✅
       - IntPtr handle = IntPtr.Zero ← ❌

## VirtualAllocEx :

a. Une fonction WinAPI qui permet d'allouer de la mémoire dans l'espace mémoire d'un autre processus
   - static extern IntPtr VirtualAllocEx(
     IntPtr hProcess, ← Handle du processus cible. (Obtenu via OpenProcess)

IntPtr lpAdress, ← Adresse mémoire souhaitée. IntPtr.Zero choisit automatiquement une adresse libre.
uint dwSize, ← Taille de la mémoire à allouer (en octets).
uint flAllocationType, ← Type d'allocation MEM_COMMIT ou MEM_RESERVE.
uint flProtect); ← Permissions de la mémoire ⚠️ Windows contrôle fortement ce paramètre.

- Retour :
  - IntPtr != IntPtr.Zero ← ✅
  - IntPtr == IntPtr.Zero ← ❌

## WriteProcessMemory :

a. Ecrit des données dans la mémoire d'un autre processus.
Donc du processus créé vers le processus cible.

- public static extern bool WriteProcessMemory(
  IntPtr hProcess, ← Handle du processus cible (Obtenu avec Openrocess)
  IntPtr lpBaseAdress, ← Adresse dans le processus cible où écrire (Obtenue via VirtualAllocEx)
  byte[] lpBuffer, ← Les données à écrire
  int nSize, ← Nombre d'octets à écrire
  outIntPtr lpNumberOfBytesWritten); ← Nombre réel d'octets écrits (Vérif que tout s'est bien passé)
  - Retour :
    - true ← ✅
    - false ← ❌

## CreateRemoteThread :

a. Une fonction WinAPI qui permet de créer un thread dans un autre processus.

- static extern IntPtr CreateRemoteThread(
  IntPtr hProcess, ← Processus cible (Obtenu via OpenProcess && doit avoir le droit PROCESS_CREATE_THREAD)
  IntPtr lpThreadAttributes, ← Attrbuts de sécurité du thread. Presque toujours IntPtr .Zero
  uint dwStackSize, ← Taille de la pile du Thread. 0 par défaut.
  IntPtr lpStartAdress, ← Adresse où le thread commence à s'exécuter.
  IntPtr lpParameter, ← Paramètre passé à la fonction du thread
  uint dwCreationFlags, ← Mode de création du thread
  out IntPtr lpThreadId); ← Reçoit l'identifiant du thread crée
  - Retour :
    - IntPtr != IntPtr.Zero ← ✅
    - IntPtr == IntPtr.Zero ← ❌

# Exercice 3 : AV Evasion

Explication de certaines fonctions et variables:

## Class Encryption :

uint cle ← Cette variable permet d'instancier une clé hexadécimale utilisée lors du chiffrement et du déchiffrement, c'est la seule variable qu'on retrouvera dans les deux parties du code, car celle-ci doit être exactement la même.

static byte[] encryptionXOR(byte[] tableauText) && static byte[] encryptionXOR(string tableauText) ← Ces fonctions sont surchargées, c'est-à-dire qu'elles font exactement la même chose mais sur des types de variable d'entrée différentes : l'une va se charger d'une liste de bytes et l'autre d'un string. La sortie de ces fonctions sera le tableau d'entrée, mais chiffré.

static void printTableauByte(string varname, byte[] data) ← Cette fonction va permettre, sur base d'un varchar et d'un tableau de byte[], de comprendre ce que la sortie de la fonction produit. Plus précisément, on ajoutera à chaque fois 2 caractères (ce qui représente un byte) et une virgule sera ajoutée à chaque élément ; une fois arrivé au dernier, on ne mettra pas de virgule.

## Class Déchiffrage :

Comme dit au-dessus, la variable cle est exactement la même que celle dans Encryption.

static byte[] xorDechiffrementBytes(byte[] chiffre) && static string xorDechiffrementBytes(byte[] chiffre) ← Ici, on va pouvoir déchiffrer un tableau de bytes, par exemple le tableau du Shellcode qui a été chiffré ^^. Une chose intéressante à savoir, c'est que c'est dans cette fonction-ci qu'on utilisera la clé utilisée dans la classe Encryption.

Main ← retourne ce qui pourrait s'apparenter à un Shellcode, mais celui-ci est chiffré avec le XOR (^) et donc ne trigger pas Windows Defender.

scBuf va prendre la taille du Shellcode chiffré.

Et au dernier moment, dans byte[] buf = xorDechiffrementBytes(buffEnc);, on va déchiffrer le Shellcode afin de ne pas activer une alerte Windows Defender.

## Retour d’expérience :

Dans un premier temps, j’étais perdu. Le premier exercice, sans la solution, je n’aurais clairement pas pu le réaliser et, sans une IA, je n’aurais pas pu comprendre comment les fonctions fonctionnaient. Non pas que les recherches sur Internet ne soient pas intéressantes, mais me retrouver sur Microsoft Learn à lire, à demi-mot, ce que fait DllImport, à quoi il sert, etc., me faisait perdre beaucoup de temps et d’envie de progresser.

Ensuite, une fois l’étape du « Mais comment je vais faire » passée, j’ai décidé de me débrouiller. Je suis allé voir chaque fonction, ce qu’est un programme managé et non managé, pourquoi et comment utiliser DllImport, pourquoi utiliser msfvenom et comment l’utiliser, d’ailleurs, WSL est une véritable pépite sur Windows.

Une fois les exercices terminés, et étant donné que je me suis obligé à me renseigner pour comprendre ce que je faisais, j’ai progressé et compris pourquoi le Shellcode est quelque chose de délicat entre de mauvaises mains. Mais en même temps, sur un ordinateur classique, une personne souhaitant effectuer des tâches automatisées pourrait s’en servir pour gérer davantage de paramètres dans son code.

Finalement, j’ai beaucoup appris sur ce qu’est un Shell Loader / Injector. J’ai apprécié chacune des erreurs que j’ai dû résoudre, car elles me rapprochaient toujours un peu plus d’une victoire et d’un code fonctionnel.