# MAINTENIR L'ACCÈS, CONTROLE ET EVASION

PYTHON POUR LA CYBERSECURITÉ

# OBJECTIFS

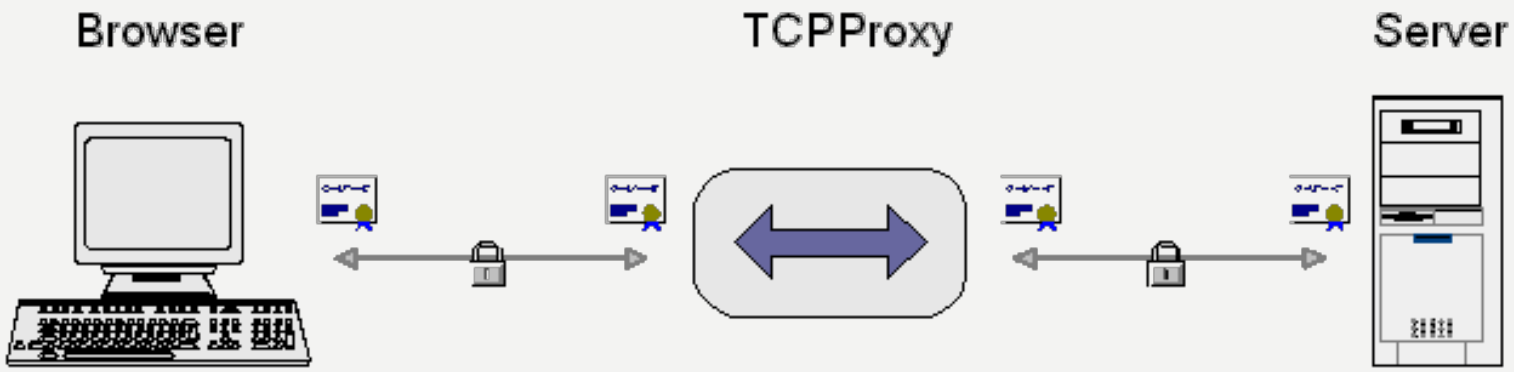
- A la fin de cette partie, vous devriez être en mesure de :
  - Décrire le concept du maintien de l'accès
  - Enumérer les méthodes principales du maintien de l'accès
  - Décrire le concept de
    - TCP proxy
    - SSH Tunneling
    - Data obfuscation
  - Ecrire un code qui permet
    - De relayer des commandes (proxy)
    - De contacter un serveur et/ou recevoir des connexions en chiffrant le trafic (SSH Tunneling)
    - De faire l'obfuscation les données dans des images

# C'EST QUOI?

- C'est le fait de pouvoir accéder à la cible par d'autres moyens que la faille exploitée à la base
  - Créer des comptes utilisateurs privilégiés
  - Installer des Trojan Horse
  - Découvrir le réseau
  - Infiltrer d'autres systèmes
- Il est important à ce stade d'enfuir le trafic malveillant dans le trafic « normal » pour leurrer les mesures de défense

# TCP PROXY

- Utile pour faire passer le trafic d'un hôte à l'autre
- Il peut être utilisé pour :
  - surveiller ou enregistrer le trafic,
  - filtrer certaines données,
  - équilibrer la charge entre plusieurs serveurs.
- Cas d'usages:
  - Impossibilité d'utiliser Wireshark
  - Impossibilité d'installer des « drivers » Windows
  - Pas de visibilité direct sur la cible



# EXERCICE

- Vous allez créer un proxy TCP en Python, capable de :
  - Accepter une connexion entrante (du client),
  - Se connecter au serveur cible,
  - Relayer les données dans les deux sens.
- Tout d'abord il faudra créer un server de test
  - Reprendre le serveur TCP fait dans les premières séances
  - Ou Télécharger le serveur de moodle
- Ensuite il faudra créer un programme qui:
  - Se lance et se met à l'écoute sur un port préconfiguré
  - Reçoit un paquet d'un client > il l'envoie à la destination et attend une réponse
  - La destination envoie la réponse > le programme renvoi la réponse au client

# EXERCICE

- Ecrire une fonction qui:
  - Prend en paramètre un paquet
  - Envoie le paquet a un socket pré-initialisé (déjà connecté à un port TCP
  - Recueille la réponse et la retourne
- Ecrire une fonction qui:
  - Crée un « socket » et se met à l'écoute sur le port 8888
  - Fait appelle à la fonction précédente et lui envoie le paquet en paramètre
  - Retourne la réponse à l'émetteur et attend un nouveau message
  - Si le message est #EXIT# ferme les connexions avec la cible et l'émetteur
- Tester le code:
  - Lancer le server TCP (port 9999)
  - Lancer le proxy (port 8888)
  - En utilisant telnet, se connecter au proxy et envoyer un message, vérifier la réception sur le serveur TCP
  - Envoyer #EXIT# et s'assurer que les connections se ferment

# SSH TUNNELING

- Paramiko: librairie python qui vous donne accès à SSH2
- SSH Tunneling: principe de faire exécuter des commandes à un client SSH
  - La cible se connecte à votre serveur SSH en utilisant votre code malveillant
  - Le serveur envoie une commande en réponse qui sera exécuté sur la cible par le code malveillant
  - La cible envoie le résultat de la commande au serveur et attend de nouvelles commandes
- Permet de contourner les pares-feux et EDR

```python
#!/usr/bin/env python
import paramiko

def ssh_command(ip, port, user, passwd, cmd):
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, port=port, username=user, password=passwd)

    _, stdout, stderr = client.exec_command(cmd)
    output = stdout.readlines() + stderr.readlines()
    if output:
        print('--- Output ---')
        for line in output:
            print(line.strip())

if __name__ == '__main__':
    import getpass
    # user = getpass.getuser()
    user = input('Username: ')
    password = getpass.getpass()

    ip = input('Enter server IP: ') or '192.168.1.203'
    port = input('Enter port or <CR>: ') or 2222
    cmd = input('Enter command or <CR>: ') or 'id'
    ssh_command(ip, port, user, password, cmd)

```

# SSH TUNNELING

- Installer paramiko
- Télécharger le script de base
- Créer 2 scripts:
  - Serveur: il va recevoir la connexion et envoyer les vraies commandes
  - Client: il va initier la connexion, recevoir les commandes, les exécuter et retourner la réponse

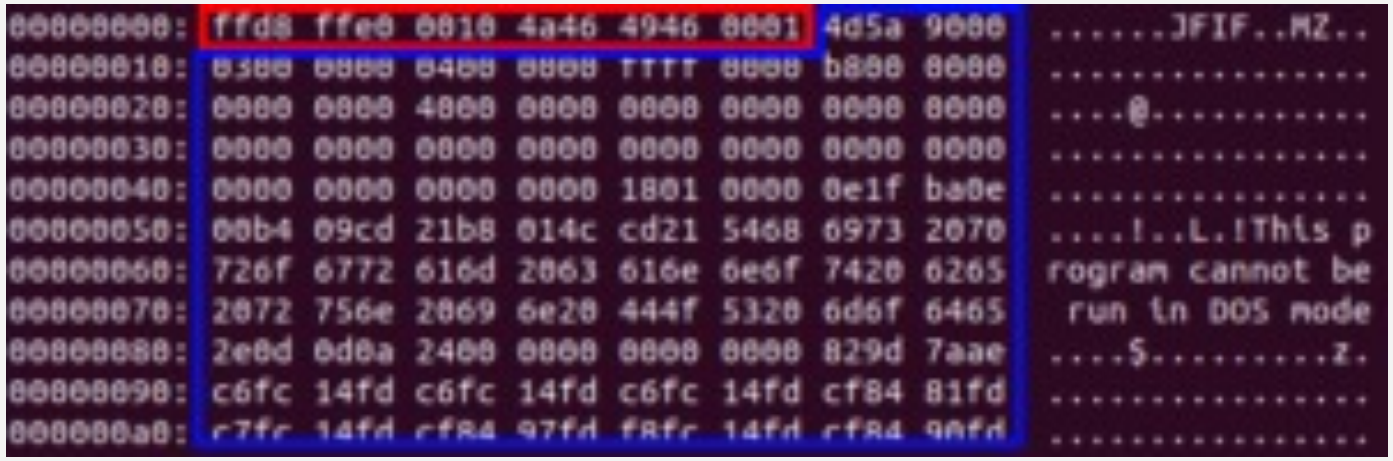
# OBFUSCATION DES DONNÉES

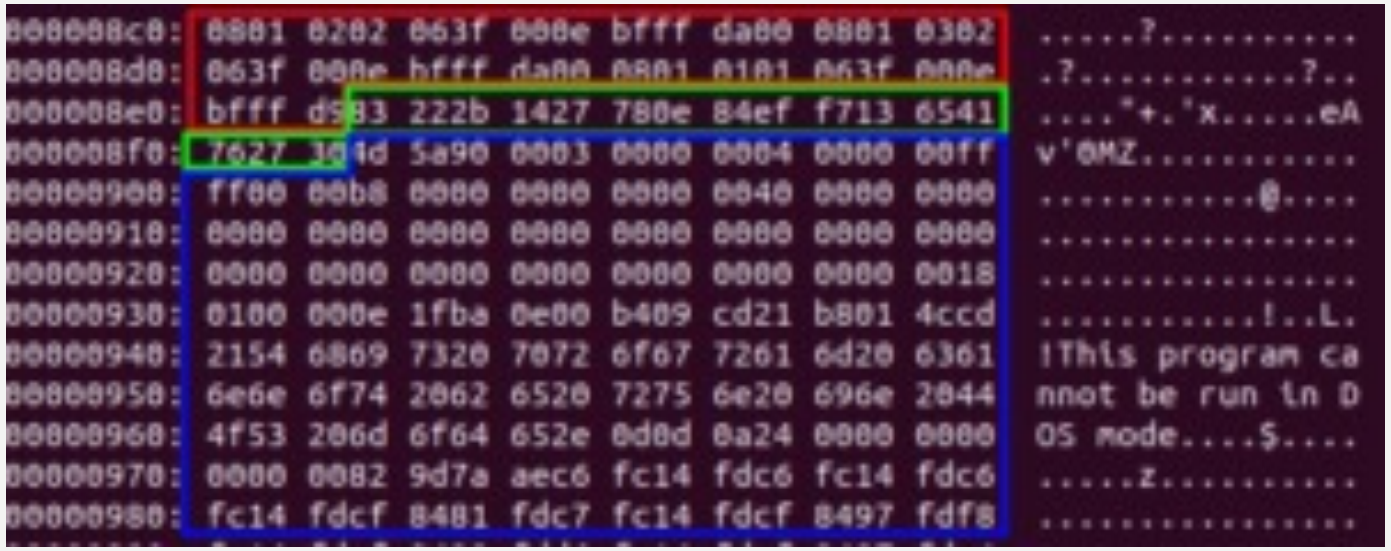
- 3 méthodes principales… bien que l'on peut imaginer plein d'autre
  - Ajout de l'entête JPEG
  - Ajout des données à une image JPEG
  - Intégration des données dans une image PNG à l'aide de la stéganographie LSB (Least Significant Byte)
- Ajout de l'entête JPEG
  - Approche naïve: création d'un fichier qui ressemble à une image JPEG
  - Simple et rapide
  - Facilement détectable surtout que visionner l'image ne donnera pas d'image

# OBFUSCATION DES DONNÉES

- Ajout des données à la fin de l'image JPEG
  - Simple et rapide
  - L'image et valide et peut être visionnée correctement
  - Pour en extraire le code, il faut connaître la taille exacte de l'image
  - Facilement détectable surtout par les EDR
- Stéganographie LSB
  - La plus complexe des 3 méthodes
  - La couleur dans les images PNG est codée sur 3 octets
  - La manipulation du dernier bit de chaque octet n'est pas détectable à l'œil nue > image valide

# OBFUSCATION DES DONNÉES



# OBFUSCATION DES DONNÉES

# OBFUSCATION DES DONNÉES

- Ouvrir le fichier image et extraire la taille pour calculer la taille maximale du code
  - from PIL import Image
  - W * H / 8
- Ouvrir le fichier de code, le lire, et le compresser
- Valider que la taille du code compressé est inférieure à la tailla maximale permise
- Utiliser le code d'obfuscation pour mettre le code dans l'image
- Sauvegarder le résultat dans une nouvelle image JPG

# OBFUSCATION DES DONNÉES - LSB

- Lecture et écriture en mode binaire car les données de l'image seront en mode binaire

```python
def read_data(input_file):
    try:
        # Ouverture du fichier en mode binaire pour lecture ('rb')
        with open(input_file, "rb") as f:
            data = f.read()  # Lecture de tout le contenu du fichier
    except IOError:
        # Gestion d'erreur : problème d'ouverture (fichier manquant, permissions, etc.)
        print("Could not open file {}".format(input_file))
        exit(1)  # Arrêt du programme avec code d'erreur
    return data  # Renvoie les données binaires lues

def write_data(output_file, data):
    try:
        # Ouverture du fichier en mode binaire pour écriture ('wb')
        with open(output_file, "wb") as f:
            f.write(data)  # Écriture des données binaires dans le fichier
    except IOError:
        # Gestion d'erreur : échec d'ouverture ou d'écriture
        print("Could not open file {}".format(output_file))
        exit(1)  # Arrêt du programme avec code d'erreur
    print("Data written to {}".format(output_file))  # Confirmation pour l'utilisateur
```

# OBFUSCATE

- Lit les données à cacher depuis un fichier (binaire ou texte).
- Convertit les données en bits, avec un entête qui encode la taille des données (en bits).
- Ouvre une image en mode lecture/écriture (format compatible avec les pixels RGB).
- Vérifie que l'image est assez grande pour contenir les données à insérer.
- Parcourt les pixels de l'image un par un :
  - Pour chaque composante R, G et B de chaque pixel :
  - Remplace le bit de poids faible (LSB) par un bit de la donnée.
  - Enregistre l'image modifiée dans un nouveau fichier (souvent en .png pour ne pas perdre les LSB).
- Affiche un message de confirmation si tout s'est bien passé.

```python
from bitstring import BitArray
from PIL import Image

LSB_PAYLOAD_LENGTH_BITS = 32
def obfuscate_via_lsb(data_file, input_file, output_file):
    # Lecture des données à cacher (fichier binaire ou texte)
    data = read_data(data_file)

    # Création de la chaîne de bits à insérer :
    # - d'abord la taille du message (en bits),
    # - puis le message lui-même encodé en binaire
    data = BitArray(uint=len(data) * 8, length=LSB_PAYLOAD_LENGTH_BITS).bin + BitArray(bytes=data).bin

    i = 0  # Compteur de position dans la chaîne de bits
    try:
        with Image.open(input_file) as img:
            width, height = img.size
            # Vérification que l'image a assez de place pour contenir les données
            if len(data) > width * height * 3:
                print("Data is too large to be embedded in the image. Data contains {} bytes, maximum is {}".format(
                    int(len(data) / 8), int(width * height * 3 / 8)))
                exit(1)

            # Parcours de chaque pixel de l'image (par colonne puis ligne)
            for x in range(0, width):
                for y in range(0, height):
                    pixel = list(img.getpixel((x, y)))  # Récupère le pixel (R, G, B)
                    for n in range(0, 3):  # Pour chaque composante (R, G, B)
                        if i < len(data):
                            # On remplace le bit de poids faible (LSB) par le bit du message
                            pixel[n] = pixel[n] & ~1 | int(data[i])
                            i += 1
                    # Mise à jour du pixel dans l'image
                    img.putpixel((x, y), tuple(pixel))

                    # Si tous les bits ont été insérés, on sort
                    if i >= len(data):
                        break
                if i >= len(data):
                    break

            # Enregistrement de l'image modifiée
            img.save(output_file, "png")
    except IOError:
        print("Could not open {}. Check that the file exists and it is a valid image file.".format(input_file))
        exit(1)

    print("Data written to {}".format(output_file))

```

# DEOBFUSCATE

- Ouvre l'image contenant les données cachées (image avec LSB modifiés).
- Lit les premiers bits (LSB des pixels) pour récupérer l'entête :
- Cette entête indique la taille du message caché (en bits).
- Lit ensuite les bits du message, en se basant sur la taille extraite.
- Assemble les bits extraits pour reformer les données d'origine :
- Conversion des bits en octets (BitArray) pour reconstruire les données binaires.
- Écrit les données extraites dans un fichier de sortie.
- Affiche un message de confirmation si tout s'est bien passé.

```python
def deobfuscate_via_lsb(input_file, output_file):
    try:
        with Image.open(input_file) as img:
            # Lecture des premiers bits qui donnent la taille du message caché
            payload_length = int("".join([str(x) for x in decode_img_nbits(img, LSB_PAYLOAD_LENGTH_BITS)]), 2)

            # Lecture de tous les bits utiles (entête + message)
            data = decode_img_nbits(img, payload_length + LSB_PAYLOAD_LENGTH_BITS)[LSB_PAYLOAD_LENGTH_BITS:]

            # Conversion des bits en données binaires (bytes)
            data = BitArray(bin="".join([str(x) for x in data])).bytes
    except IOError:
        print("Could not open {}".format(input_file))
        exit(1)

    # Écriture des données extraites dans le fichier de sortie
    write_data(output_file, data)


def decode_img_nbits(img, nbits):
    data = []
    i = 0
    width, height = img.size
    # Parcours de tous les pixels pour extraire les LSB
    for x in range(0, width):
        for y in range(0, height):
            pixel = list(img.getpixel((x, y)))
            for n in range(0, 3):  # Pour chaque composante (R, G, B)
                if i < nbits:
                    # On extrait le bit de poids faible de chaque composante
                    data.append(pixel[n] & 1)
                    i += 1
            if i >= nbits:
                break
        if i >= nbits:
            break
    return data
```